Universidad Autónoma de Madrid
Escuela politécnica superior

Universidad Autónoma de Madrid

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

SISTEMA DE CLASIFICACIÓN DE PAQUETES A ALTA TASA UTILIZANDO REDES NEURONALES CONVOLUCIONALES Y FPGAS

Autor: Víctor Morales Gómez
Tutores: Jorge Enrique López de Vergara Méndez
y Gustavo Daniel Sutter Capristo

JUNIO 2021

SISTEMA DE CLASIFICACIÓN DE PAQUETES A ALTA TASA UTILIZANDO REDES NEURONALES CONVOLUCIONALES Y FPGAS

Autor: Víctor Morales Gómez
Tutores: Jorge Enrique López de Vergara Méndez
y Gustavo Daniel Sutter Capristo

High Performance Computing and Networking Research Group
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
JUNIO 2021

Resumen

Hoy en día, la clasificación del tráfico de red se ha convertido en una tarea fundamental, con aplicaciones tan variadas como aplicar distintas clases de servicio o identificar malware. Sin embargo, las técnicas previamente existentes, tales como los números de puerto o la inspección profunda de paquetes, ya no son válidas, pues un porcentaje amplio del tráfico de Internet va cifrado, lo que dificulta su clasificación.

En este Trabajo Fin de Máster se plantean aplicar redes neuronales convolucionales y recurrentes para identificar y clasificar distintos tipos de tráfico cifrado. Para ello, se partirá de conjuntos de datos públicos que contienen paquetes cifrados etiquetados, los cuales deberán tratarse como imágenes, separando los paquetes de cada flujo, para evitar que la red pueda distinguir un paquete al conocer otro del mismo flujo. Las imágenes se generan quitando cabeceras de nivel de enlace, red y transporte. Se utilizará un subconjunto de los flujos para entrenar con sus paquetes la red neuronal, estudiándose el rendimiento de la solución con el resto del conjunto de datos en términos de precisión, sensibilidad, exhaustividad, etc.

Como complemento a lo anterior, se estudiará adicionalmente la implementación de este tipo de redes neuronales en una FPGA, de manera que pueda procesarse el tráfico de red en tiempo real y a alta tasa. Para ello, se aprovecharán las unidades de procesamiento DPU disponibles en las últimas arquitecturas FPGA. Esto permitirá llevar los estudios teóricos existentes a una aplicación de los mismos en un entorno más realista.

Palabras Clave

Análisis, cifrado, clasificación, DPU, flujo, FPGA, identificación, Pynq, redes neuronales, Tensorflow.

Abstract

Nowadays, the classification of network traffic has become a fundamental task, with applications as varied as enforcing different classes of service or identifying malware. However, previously existing techniques, such as port numbers and even deep packet inspection, are no longer valid, as a large percentage of Internet traffic is encrypted, making it difficult to classify.

In this Master's Thesis, convolutional and recurrent neural networks will be applied to identify and classify different types of encrypted traffic. To do so, we will start from public data sets containing labelled encrypted packets, which must be treated as images, separating the packets of each flow, to prevent the network from being able to distinguish a packet when it learns about another packet of the same flow. Images may be generated by removing link, network and transport level headers. A subset of the flows will be used to train the neural network with their packets, and the performance of the solution will be studied with the rest of the dataset in terms of accuracy, recall, f-score, etc.

As a complement to the above, the implementation of this type of neural networks in an FPGA will also be studied, so that the network traffic can be processed in real time and at a high rate. For this purpose, the DPU processing units available in the latest FPGA architectures will be used. This will allow existing theoretical studies to be taken to an application in a more realistic environment.

Keywords

Analysis, cypher, classification, DPU, flow, FPGA, identification, , Pynq, neural network, Tensorflow.

Agradecimientos

Me gustaría comenzar agradeciendo a mis padres. A Silvia, mi madre, por ser una fuente inagotable de amor y cariño, así como un excelente ejemplo de trabajo y dedicación. También a mi padre, Antonio, porque un día decidiste regalarme un libro, El Diablo de los Números, sin saber que despertaría en mí una gran pasión por las matemáticas. Eres mi ejemplo a seguir y aunque no siempre estemos de acuerdo, agradezco todos y cada uno de los consejos que me proporcionas.

También quiero agradecer a mi hermano Carlos, que siguiendo los pasos de esta familia has decidido estudiar Teleco, eres una mente prodigiosa y te espera un futuro brillante. No me olvido de Dexter, nuestro perro, a él también quiero agradecerle el amor incondicional que siempre nos aporta.

A su vez deseo agradecer a todos mis amigos el tiempo que pasamos juntos. Y también a mis compañeros de la carrera. Que juntos hemos sobrevivido a los exámenes de CAP, ondas y antenas. En concreto, quiero agradecer a aquellos con quienes sé que voy a mantener una amistad duradera. Adri, Alex, Belén, David, Ernesto, Ferchu, Javi, Patri y a todo el Agrupamiento Primario.

Finalmente, quiero agradecer a todos los profesionales de la comunidad universitaria por el gran trabajo que realizan. Al grupo HPCN, a Franco Capraro y a mis tutores Jorge Enrique López de Vergara Méndez y Gustavo Daniel Sutter Capristo por proponerme este trabajo del que he aprendido muchísimo.

¡Muchas gracias a todos!

Víctor Morales Gómez

Junio 2021

Índice general

Índice de Figuras

Índice de Tablas

Glosario de acrónimos

- **ACK**: *Acknowledgement*, Asentimiento.
- **AI**: *Artificial Intelligence*, Inteligencia Artificial.
- **ARP**: *Address Resolution Protocol*, Protocolo de Resolución de Direcciones
- **CNN**: *Convolutional Neural Network*, Red Neuronal Convolucional.
- **CPU**: *Central Processing Unit*, Unidad de Procesamiento Central.
- **DHCP**: *Dynamic Host Configuration Protocol*, Protocolo de Configuración Dinámica de Equipos.
- **DNS**: *Domain Name System*, Sistema de Nombres de Dominio.
- **DoH**: *DNS over HTTPS*, DNS sobre HTTPS.
- **DPU**: *Deep Learning Processing Unit*, Unidad de Procesamiento de Aprendizaje Profundo.
- **FPGA**: *Field Programmable Gate Array*, Matriz de Puertas Lógicas Programable en Campo.
- **GPU**: *Graphic Processing Unit*, Unidad de Procesamiento Gráfico.
- **HDL**: *Hardware Description Language*, Lenguajes de Descripción de Hardware
- **HTTP**: *Hypertext Transfer Protocol*, Protocolo de Transferencia de Hipertexto.
- **IO**: *Input-Output*, Entrada-Salida.
- **IP**: *Internet Protocol*, Protocolo de Internet.
- **ISP**: *Internet Service Provider*, Proveedor de Servicios de Internet.
- **LSTM**: *Long Short Term Memory*, Redes de Memoria a Largo y Corto plazo.
- **MPSoC**: *Multi-processor System-On-Chip*, Sistema en Chip Multiprocesador
- **MTU**: *Maximum Transmission Unit*, Unidad Máxima de Transferencia
- **NTP**: *Network Time Protocol*, Protocolo de Tiempo en la Red
- **QoE**: *Quality of Experience*, Calidad de Experiencia.

- **QoS**: *Quality of Service*, Calidad de Servicio.
- **QUIC**: *Quick UDP Internet Connections*, Conexiones UDP Rápidas en Internet.
- **RNN**: *Recurrent Neural Network*, Red Neuronal Recurrente.
- **RTT**: *Round-Trip Time*, Tiempo de Ida y Vuelta.
- **RTL**: *Register Transfer Level*, Lenguaje de Transferencia de Registros.
- **SNMP**: *Deep Packet Inspection*, Inspección Profunda de Paquetes
- **SNMP**: *Simple Network Management Protocol*, Protocolo Simple de Gestión de Red.
- **SoC**: *System-On-Chip*, Sistema en Chip
- **SSL**: *Secure Sockets Layer*, Capa de *Sockets* Seguros.
- **TCP**: *Transmission Control Protocol*, Protocolo de Control de Transmisiones.
- **TLS**: *Transport Layer Security*, Seguridad de la Capa de Transporte.
- **UDP**: *User Datagram Protocol*, Protocolo de Datagramas de Usuario.
- **URL**: *Uniform Resource Locator*, Localizador de Recursos Uniformes
- **VoIP**: *Voice Over IP*, Voz sobre Protocolo de Internet.
- **VPN**: *Virtual Private Network*, Red Privada Virtual

1 Introducción

1.1. Motivación

En los últimos años se ha puesto cada vez más esfuerzos en la seguridad y protección de los sistemas informáticos y la privacidad de los datos de sus usuarios. Para ello, se usan protocolos de cifrado de datos que evitan que la información enviada por Internet pueda ser revelada por terceros. Sin embargo, esto crea dificultades a los ISP (*Internet Service Provider*, Proveedor de Servicios de Internet) para poder gestionar sus sistemas de forma eficiente. Esto se debe a que el cifrado de los paquetes de información evita la clasificación de estos según su aplicación. [13] La clasificación del tráfico es un pilar fundamental en el dimensionado, optimización, despliegue, medición, administración y securización de las redes.

Las técnicas previamente existentes para la clasificación del tráfico incluían el estudio de puertos, inspección profunda de paquetes o uso de estadísticos del flujo. Las primeras técnicas no pueden ser realizadas en paquetes de tráfico cifrado y la última tiene la desventaja de que aumenta la latencia de procesamiento debido a que requiere la llegada de varios paquetes del mismo flujo para poder procesarlo. Por lo tanto, se dará un mal servicio a las aplicaciones críticas en latencia como VoIP (*Voice Over IP, Voz sobre Protocolo de Internet*), videollamadas y videojuegos en línea. [19]

Para resolver los inconvenientes de las técnicas anteriores, en este trabajo se plantea el uso de redes neuronales convolucionales y recurrentes para identificar y clasificar distintos tipos de tráfico cifrado. Estas redes permiten que una inteligencia artificial pueda aprender a clasificar tráfico cifrado a partir de un conjunto de datos etiquetado. [17] También se propone el uso de FPGAs (*Field Programmable Gate Array, Matriz de Puertas Lógicas Programable en Campo.*) para aprovechar las unidades de procesamiento DPU (*Deep Learning Processing Unit, Unidad de Procesamiento de Aprendizaje Profundo*) que se incorporan en las arquitecturas más recientes. [12] Esto podría permitir clasificar el tráfico de red en tiempo real y a alta tasa, llevando los estudios teóricos a una aplicación en un entorno más realista.

1.2. Objetivos

El objetivo de este Trabajo de Fin de Máster es desarrollar un sistema de clasificación de tráfico cifrado mediante el uso de redes neuronales convolucionales y la utilización de FPGAs para procesar el sistema de clasificación en redes de alta tasa.

Este objetivo general podrá alcanzarse si se cumplen los siguientes subobjetivos:

- Estudiar las redes neuronales convolucionales y recurrentes aplicadas al problema de la clasificación de tráfico de red cifrado.
- Obtener un conjunto de datos públicos con paquetes cifrados etiquetados.
- Desarrollar un sistema de procesamiento del tráfico cifrado para convertir los paquetes en imágenes.
- Aprender a diseñar y desarrollar un sistema de clasificación de tráfico cifrado utilizando las redes neuronales.
- Validar la efectividad del sistema de clasificación desarrollado.
- Comprender el funcionamiento de los sistemas DPU de las FPGAs.
- Analizar las ventajas e inconvenientes asociados a la implementación de redes neuronales en sistemas FPGAs.

1.3. Fases de realización

La realización de este Trabajo de Fin de Máster se ha llevado a cabo siguiendo la siguiente planificación de horas, como se puede ver en el diagrama de Gantt de la **Figura 1.1**:

- **Estudio del estado del arte:** En esta parte del Trabajo de Fin de Máster, se estudiaron artículos, libros y guías, para obtener el conocimiento necesario para acometer el desarrollo posterior. Principalmente, se estudiaron los sistemas de clasificación de tráfico clásicos: mediante puertos bien conocidos, inspección de paquete profunda y análisis de estadísticos del flujo de red. [13] Pero también los nuevos métodos de clasificación basados en redes neuronales. Por último, se estudió la guía de Xilinx Vitis AI para utilizar la biblioteca *TensorFlow* y DPU para inferencia en FPGAs. [11] [20]
- **Diseño y Desarrollo:** Durante esta tarea, se descargó un conjunto de datos de tráfico cifrado etiquetado, este se procesó para obtener imágenes unidimensionales y bidimensionales de los paquetes y se organizaron según los flujos de red a los que pertenecían los paquetes. Luego, se implementó una red neuronal convolucional y recurrente para clasificar las imágenes. Por último, este sistema se procesó con Vitis AI para obtener un fichero ejecutable en el sistema DPU de una FPGA-ZCU104.

Tareas / subtareas	Horas
T1. Estudio del Estado del Arte	**50**
T1.1 Estudio de los diferentes métodos para la clasificación de tráfico.	10
T1.2 Estudio de los sistemas de redes neuronales convolucionales y recurrentes aplicados al problema de la clasificación de tráfico cifrado.	25
T 1.3 Estudio de la documentación de Vitis AI de Xilinx para implementar redes neuronales en sistemas FPGA con DPU.	15
T2. Diseño y Desarrollo	**150**
T2.1 Descarga, estudio y filtrado del dataset de tráfico cifrado etiquetado.	20
T2.2 Desarrollar sistema para convertir paquetes de tráfico cifrado en imágenes en 1D y 2D.	20
T2.3 Diseñar y desarrollar un sistema de clasificación de tráfico cifrado utilizando redes neuronales convolucionales y recurrentes.	60
T2.4 Implementar el modelo de red neuronal entrenado en una FPGA con DPU. Utilizando para ello la herramienta Vitis AI y el sistema Pynq.	50
T3. Validación	**70**
T3.1 Obtención de métricas de rendimiento para la red neuronal.	20
T3.2 Obtención de métricas de rendimiento para el sistema FPGA.	20
T3.3 Comparación de los resultados con soluciones existentes.	20
T3.4 Extracción de conclusiones.	10
T4. Documentación	**30**
T4.1 Escribir Memoria Final.	20
T4.2 Preparar presentación y defensa	10
TOTAL HORAS	**300**

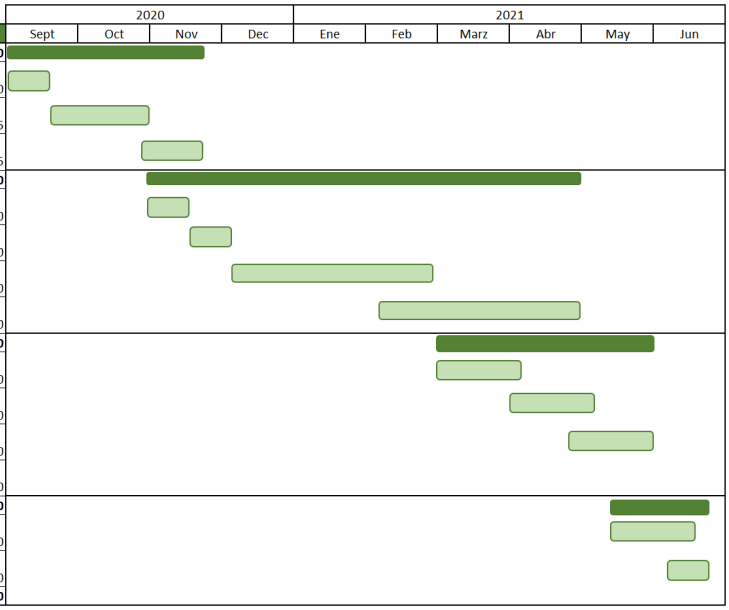

Figura 1.1: Diagrama de Gantt.

- **Pruebas y resultados:** En este apartado se comprobó el correcto funcionamiento de todo el sistema. Se han realizado pruebas de rendimiento, en cuanto a precisión del modelo y de tasa de procesamiento en la FPGA. También se comprobó la clasificación del tráfico según las distintas clases. Por último, se llevaron a cabo pruebas de rendimiento y validación cruzada entre la DPU y el sistema CPU (*Graphic Processing Unit, Unidad de Procesamiento Gráfico.*) + GPU (*Graphic Processing Unit, Unidad de Procesamiento Gráfico.*).

- **Escritura de la memoria:** Por último, se llevó a cabo la redacción de la memoria del Trabajo Fin de Máster. En ella se describe todo lo aprendido durante este proceso y la información relevante en caso de que un futuro se decida proseguir la investigación de este desarrollo.

1.4. Estructura del documento

El resto de la memoria consta de los siguientes capítulos:

- **Estado del Arte:** En este capítulo se explicarán los diferentes métodos de clasificación de tráfico, sus ventajas e inconvenientes. Luego se explicará por qué los métodos clásicos no pueden utilizarse con tráfico cifrado en entornos reales de alta tasa y baja latencia. Se explicará como la literatura actual resuelve dicho problema mediante el uso de redes neuronales convolucionales y recurrentes y las ventajas y desventajas que presenta dicho método de clasificación. También se explicarán qué son y cómo funcionan las redes neuronales convolucionales y recurrentes. Se comentará el entorno de desarrollo utilizado, así como el lenguaje de programación y las bibliotecas utilizadas. Por último, se comentarán qué sistemas existen para la inferencia de redes neuronales en FPGAs y qué es Xilinx Vitis AI. Se describirán

conceptos como la cuantización y compilación de modelos para generar archivos ejecutables en sistemas DPU como Pynq.

- **Análisis de Requisitos:** En esta parte se definirán los requisitos que debe cumplir el clasificador de tráfico para ser válido en resolver el problema propuesto. Se deberán considerar los factores de precisión, rendimiento y latencia. También se expondrán los requisitos no funcionales que debería cumplir para trabajar a alta tasa. Por último, se describe el entorno de desarrollo utilizado y el flujo de trabajo seguido.

- **Diseño:** En este apartado se explicarán las decisiones y pasos que se han seguido para diseñar el sistema de clasificación de tráfico. También se explicará el conjunto de datos utilizados y por qué la elección de este *dataset*.

- **Desarrollo:** En este capítulo se explica el desarrollo del sistema de procesamiento del *dataset* para adaptarlo a imágenes por flujos. Con estas imágenes se podrá entrenar y testear una red neuronal convolucional y recurrente, que aprenda a clasificar las imágenes de paquetes de tráfico según las características específicas de estas y según los flujos de red. Por último, se desarrollará un sistema para usar este modelo de red entrenada en una FPGA, para ello, se utilizará el entorno de desarrollo Vitis AI.

- **Pruebas y Resultados:** En esta fase se mostrarán los resultados obtenidos por el clasificador usando un subconjunto de muestras de los flujos. Se estudiará el rendimiento en términos de precisión, sensibilidad y exhaustividad. También se pondrá énfasis en los valores de velocidad de procesamiento y latencia, tanto para el sistema CPU + GPU como para el sistema DPU.

- **Conclusiones y Trabajo Futuro:** Por último, se explicará las conclusiones finales obtenidas al terminar el desarrollo del Trabajo de Fin de Máster. También se identificarán nuevas líneas de investigación basadas en el clasificador de tráfico y posibles trabajos según los resultados obtenidos del sistema DPU.

2

Estado del Arte

2.1. Introducción

En el capítulo anterior se han explicado la necesidad de realizar este trabajo. También se ha determinado los objetivos que se deben lograr para completar el Trabajo Fin de Máster. Por último, se han comentado las fases de realización del proyecto y la estructura de la memoria. Con esta información se puede afrontar la realización del proyecto.

Sin embargo, antes de comenzar con el desarrollo se expondrán los conocimientos previos necesarios a la realización del proyecto. En este capítulo se pondrá en contexto la situación en la que se encuentran los clasificadores de tráfico de red, sus características y sus limitaciones.

A continuación se profundizará en la solución que ofrece el aprendizaje automático en lo que se refiere a clasificar tráfico cifrado. En este mismo apartado se presentarán las redes neuronales convolucionales y recurrentes, que son actualmente utilizadas para resolver el problema. Finalmente se hará una revisión de los conocimientos de hardware necesarios para llevar a cabo la ejecución de las redes neuronales en dispositivos FPGAs.

2.2. Clasificación de tráfico de red

La RFC 2475 [21] define la clasificación de tráfico como: "El proceso automático que permite categorizar el tráfico de redes de computadoras de acuerdo a unos parámetros en diferentes clases de tráfico". Este proceso resulta crucial para la gestión de redes en la actualidad, ya que permite al gestor de red obtener la información necesaria para asegurar que se está dando un servicio correcto. La clasificación de tráfico de red es la base para la gestión de redes en las siguientes áreas: [13]

- **Dimensionado de redes:** El dimensionado de red consiste en las técnicas y conocimientos necesarios para llevar a cabo la correcta planificación de una solución

específica. Por ello, sabiendo el tipo de tráfico que circula la red, se podrá configurar para su buen funcionamiento. Podría servir tanto para ampliar redes existentes, como para diseñar nuevas redes.

- **Optimización de recursos:** Una red en funcionamiento es difícil de actualizar. Es por ello que se deben optimizar los recursos existentes para maximizar la calidad de experiencia de uso de los usuarios. Realizando mediciones y clasificando tráfico se podrán establecer rutas prioritarias o reservar sistemas de redundancia para las aplicaciones críticas.

- **Estrategias de negocio:** Si bien un ISP debería respetar el principio de neutralidad de la red, también podría tener intereses especiales en promocionar sus soluciones específicas de software. Por ello, detectar los servicios de empresas competidoras y dar una prioridad alta al tráfico propio podría ayudar a mejorar las ventas. Sin entrar en la cuestión ética de dicha táctica empresarial, hay recientes casos de uso de estas estrategias. En la actualidad lo más común son los acuerdos de integración de contenidos, para acercarlos a los usuarios y reducir la carga en la red.[22]

- **Seguridad:** La seguridad de las redes privadas siempre ha resultado crucial para el buen funcionamiento de los sistemas. Además, en un mundo cada vez más conectado esta tarea se ha convertido en fundamental. Detectar tráfico malicioso o intrusiones en el sistema debe ser una medida pasiva y activa llevada a cabo por los gestores de redes.

- **Diferenciación de servicios:** Desde el punto de vista de un ISP, la diferenciación de servicios resulta crucial para el funcionamiento de la red. Permite modular los recursos que se ofrecen a cada usuario según las necesidades que tenga su aplicación. De forma que, independientemente de las medidas de QoS (*Quality of Service*, Calidad de Servicio) se obtienen buenos resultados de QoE (*Quality of Experience*, Calidad de Experiencia), que se reflejarán en mayores ventas y beneficios empresariales. Otra opción posible sería un gestor de red privada de alguna empresa, cuya política dictamine que las aplicaciones de redes sociales deban ser bloqueadas para evitar distracciones o pérdida de productividad.

  Por ejemplo, recibir un correo 2 segundos después de que se haya enviado no resulta en una pérdida de calidad de experiencia. Sin embargo, realizar una llamada VoIP con 2 segundos de latencia resulta una muy mala experiencia. Otro ejemplo posible es que una descarga de vídeo en *streaming* será mejor valorada si tiene poco *jitter* aunque tenga poco ancho de banda, al contrario que una descarga de archivos masivos, donde el ancho de banda independientemente del *jitter* es preferible. Por ello, un ISP debería de priorizar el tráfico VoIP o vídeo *streaming* para minimizar la latencia del servicio y el *jitter*.

El tráfico de red puede ser clasificado según distintos parámetros, como pueden ser: según la característica de tráfico que envían (correo, web, llamada ...), otra opción sería según el tipo de aplicación o servicio (Youtube, Skype, HTTP, Gmail). También se puede clasificar según otros tipos de características específicas de tráfico (tiempo real como videollamadas o videojuegos online o tráfico masivo, como las descargas de ficheros o envío de mensajes y correos). Por último, cabe pensar que es mucho más simple identificar un

único servicio de tráfico en vez de tener que clasificar entre varias clases distintas. Esto puede resultar útil si, por ejemplo, solo deseamos priorizar las llamadas de VoIP. En el campo del aprendizaje automático se denomina a este problema: clasificación biclase frente a la clasificación multiclase.

Una vez se han visto los motivos por lo que la clasificación de tráfico resulta una tarea tan importante, se van a describir los sistemas tradicionales de clasificación de tráfico. Estos se utilizaban en tráfico de red no cifrado, en la actualidad los datos deben ser protegidos por lo que muchos de estos sistemas tienen un uso específico y limitado.[23]

2.2.1. Número de puerto

Los puertos TCP y UDP del rango de 0 a 1023 son los llamados puertos bien conocidos; estos se gestionan por IANA y son un estándar de uso de diferentes servicios de red. Los puertos hasta 49151 son los puertos reservados; aquellos servicios que deseen utilizar un puerto en específico podrían reservar uno dentro de ese rango, una vez dada la aprobación de la entidad competente.

De cara a la clasificación de tráfico, este sistema basado en puertos supone una gran ventaja, ya que cualquier conexión asociada a alguno de estos puertos podría ser identificada fácilmente. Esto permite clasificar aplicaciones como DNS (*Domain Name System*, Sistema de Nombres de Dominio), DHCP (*Dynamic Host Configuration Protocol*, Protocolo de Configuración Dinámica de *Host*) o SNMP (*Simple Network Management Protocol*, Protocolo Simple de Gestión de Red) de forma ligera y fiable. Algunos ejemplos de los puertos conocidos más importantes son los de la **Tabla 2.1**.

Tabla 2.1: Puertos bien conocidos y aplicaciones. [16]

Port Number	Protocol
20, 21	File Transfer Protocol (FTP)
22	Secure Shell (SSH)
23	Telnet Protocol
25	Simple Mail Transfer Protocol (SMTP)
53	Domain Name System (DNS)
67, 68	Dynamic Host Configuration Protocol (DHCP)
80	HyperText Transfer Protocol (HTTP)
110	Post Office Protocol (POP3)
137	NetBIOS Name Service
143	Internet Message Access Protocol (IMAP4)
443	Secure HTTP (HTTPS)
445	Microsoft-DS (Active Directory)

Sin embargo, hay aplicaciones que no desean ser clasificadas. Pensemos, por ejemplo, en aplicaciones como Skype o BitTorrent. Si tuviesen un puerto fijo podrían ser bloqueadas rápidamente, bien porque son competencia directa de otros productos empresariales propios del ISP (como el caso de Skype y las tarifas de telefonía móvil), porque la difusión de contenido mediante descargas se considera ilegal (BiTorrent y los mp3 de música) o por cuestiones de privacidad de la información del usuario. En estos casos, las aplicaciones hacen uso de los puertos efímeros o dinámicos en el rango 49152 al 65535. Estos puertos, al ser variables no son bien conocidos y por lo tanto pueden ser clasificados por este método.

2.2.2. Inspección profunda de paquetes

La inspección profunda de paquetes o DPI *Deep Packet Inspection, Inspección Profunda de Paquetes* son las técnicas utilizadas para inspeccionar la carga útil del paquete con el objetivo de encontrar información específica de alguna aplicación. Como se observa en la **Figura 2.1** se puede utilizar para identificar aplicaciones, servicios, equipos o software malicioso.

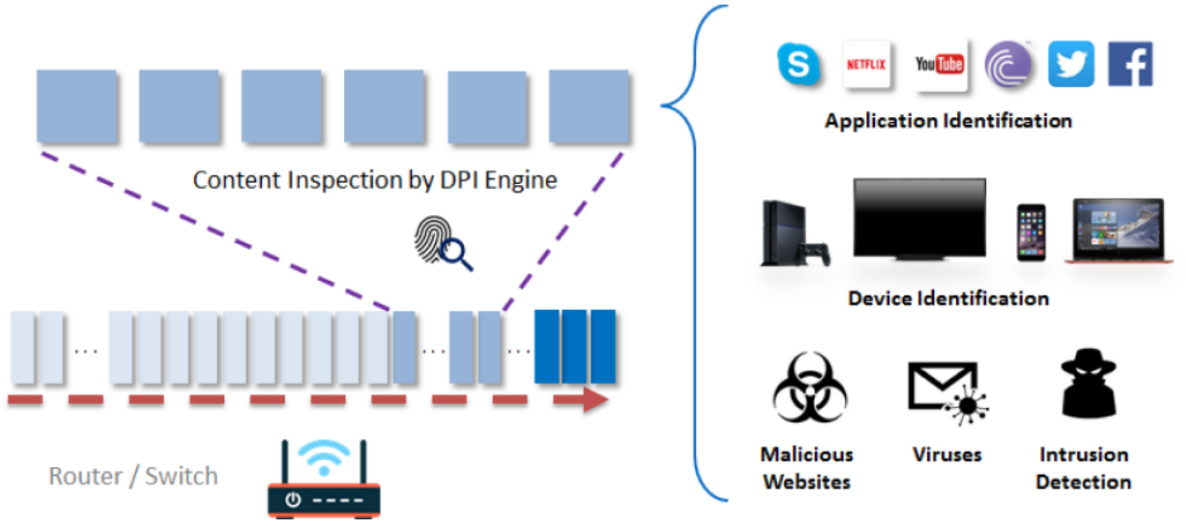

Figura 2.1: Clasificación de aplicaciones mediante DPI. [1]

Para realizar esta técnica se requieren conocimientos previos sobre el protocolo o aplicación a identificar. Para ello, se utilizan las firmas, como por ejemplo, "0x13 Bittorrent protocol", o en HTTP/1 GET: "host: www.elpais.com". Si en un paquete se encuentra alguna de las siguientes cadenas es bastante probable que sea un paquete de tráfico BitTorrent o una petición a la web *elpais.com*. En la **Figura 2.2** se observa la estructura de cabecera de los protocolos y que en el *user data* del nivel de aplicación es dónde se deben buscar los datos en un sistema DPI.

Este sistema presenta algunas desventajas: la primera es que requiere de control y evaluación de los resultados mediante un sistema de *Ground Truth*, evaluando los falsos positivos y falsos negativos. Otro problema es la alta carga computacional que presenta. En los enlaces grandes de más de 10 Gbps resulta muy complicado estudiar todos los paquetes en profundidad. Se deben tomar decisiones para el correcto funcionamiento. [24]

Una opción es establecer la búsqueda de las firmas únicamente en los primeros paquetes del flujo, pues previsiblemente una aplicación negociará y se configurará al inicio de la sesión. También se debe limitar cuántos bytes por paquete se analizan; una buena solución es estudiar los primeros y últimos 100 bytes [25], ya que es ahí donde se suelen encontrar los datos de cabeceras e información negociada en la conexión. También presenta dudas sobre como se trata la información del usuario y el derecho a la privacidad en las comunicaciones. Otro gran problema de los sistemas DPI, es que resultan muy limitados por el cifrado de los paquetes, pues al hacer ilegible la información enviada resulta muy complicado encontrar patrones generales.

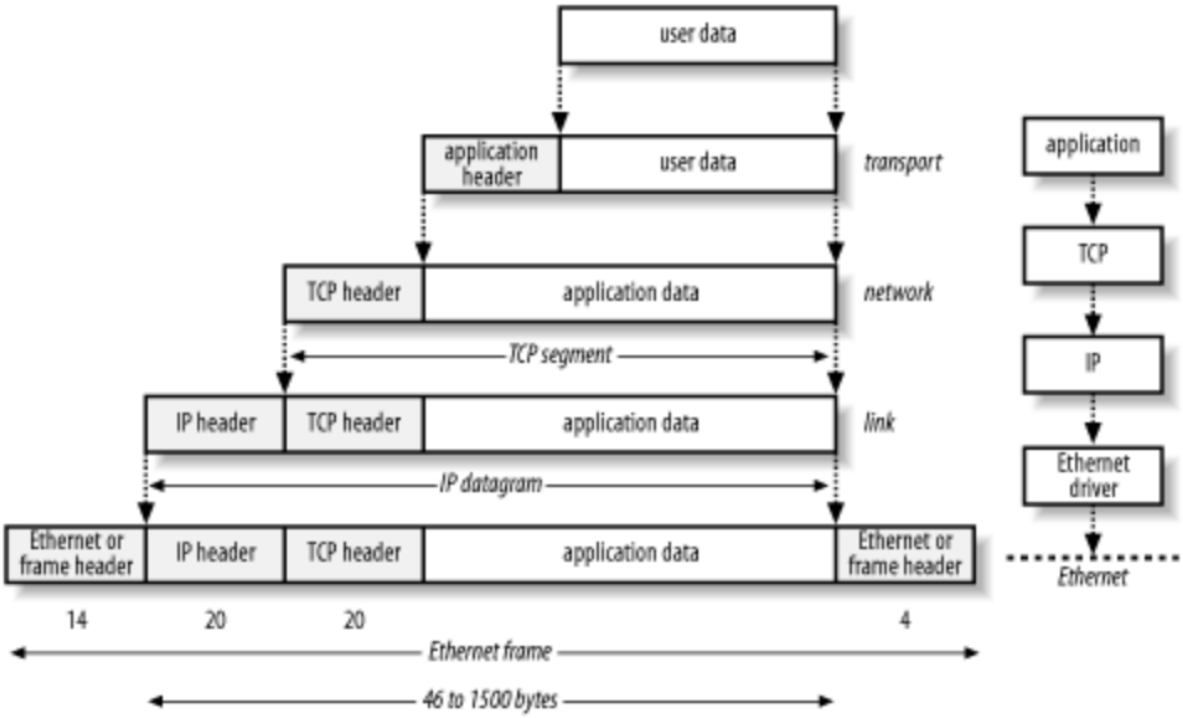

Figura 2.2: Cabecera de protocolos típica en TCP/IP. [2]

Por último, comentar que uno de los grandes problemas de los sistemas DPI en redes modernas es la latencia que introducen. Si bien la clasificación en altas tasas de ancho de banda es un reto computacional considerable, realizar esta clasificación sin añadir latencia perceptible a los servicios identificados es un reto aún mayor. [24]

Consideremos, por ejemplo, una llamada VoIP. Por lo general, se establece en los 400 ms de latencia el límite máximo para considerar un mal servicio, pero es que se debe intentar tener una latencia menor de 150 ms, que es la mínima detectable por el ser humano. [26] De estos 150 ms, más de 100 ms se pueden ir únicamente en el envío físico de la información mediante fibra óptica transcontinental. De los 50 ms que nos quedan, debemos considerar enrutamiento, codificación y decodificación, buffer de recepción y posibles retenciones en enlaces saturados. Esto dejaría el margen de la clasificación mediante DPI en una decena de ms o, idealmente, menor a un ms. Esta velocidad de procesamiento puede ser difícil de alcanzar y es un problema que veremos repetido en el siguiente apartado.

2.2.3. Estadísticas de flujo de red

Como se ha visto en el apartado anterior, la clasificación de tráfico cifrado impide usar las características "individuales" del paquete. Pero eso no significa que las características de los flujos deban ser desechadas inmediatamente. No se puede ignorar que la información enviada por la red tiene su origen en alguna aplicación, o algoritmo. Este, a su vez, deberá respetar ciertas características y reglas en su comportamiento. Ser capaces de identificar estos mecanismos de generación y envío de información puede ayudar a clasificar el tráfico.

Es por eso que estudiar las estadísticas de los flujos de red según la aplicación utilizada puede utilizarse como método de clasificación de tráfico. Las características más usadas

son el tamaño medio del paquete, el tiempo entre llegadas de paquetes, o su tasa. Estos valores se pueden utilizar en técnicas de aprendizaje automático (o *machine learning*), como los árboles de decisión o los algoritmos agrupadores de clases. Por ejemplo, en la **Figura 2.3** se observa como distintas aplicaciones sobre HTTP entre distintos tipos de servidores pueden tener diferentes distribuciones de longitud de paquete.

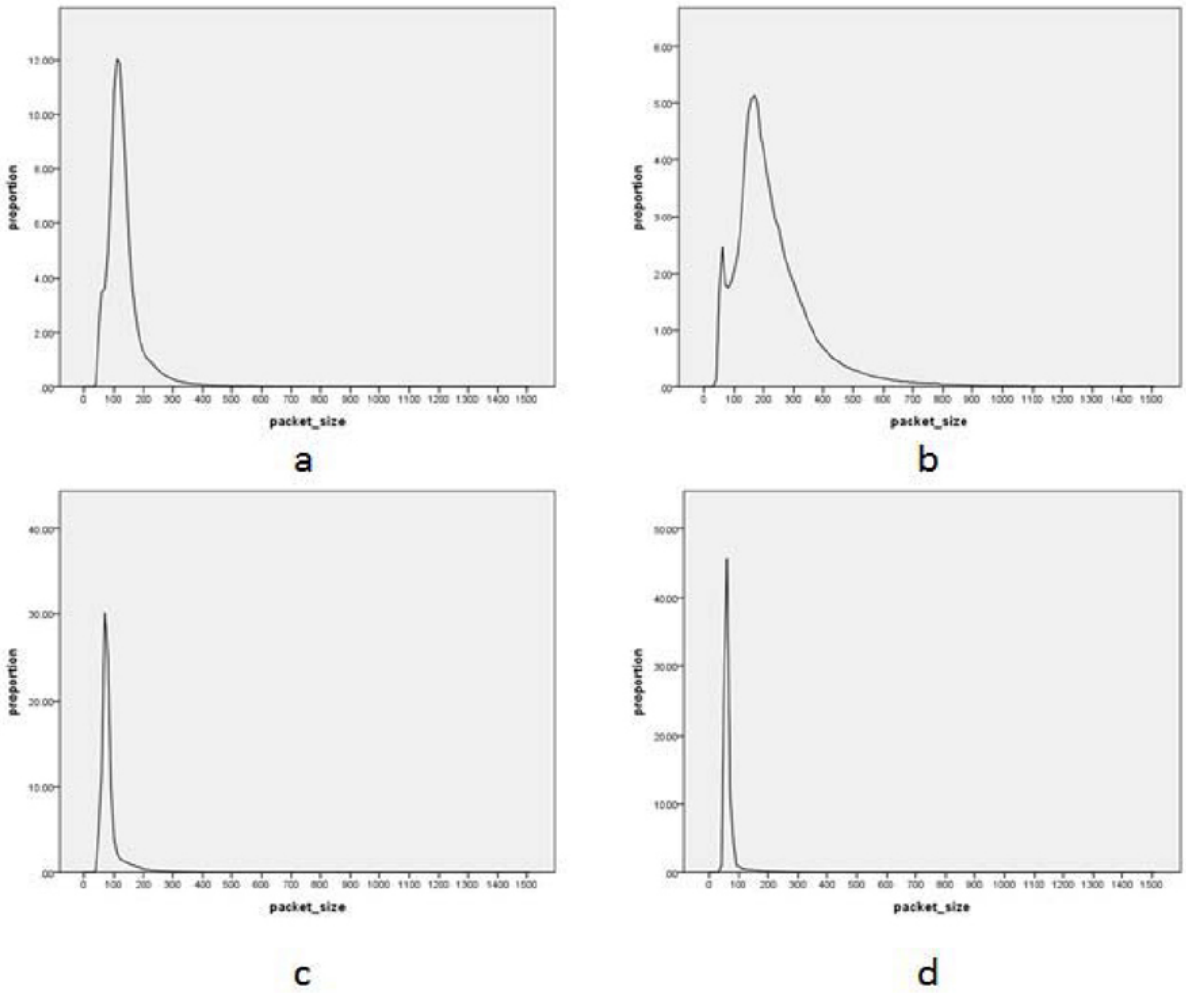

Figura 2.3: Distribución del tamaño de paquetes de HTTP. [3]

La precisión por aplicación puede ser limitada, pero el uso de patrones de comportamiento puede ayudar a clasificar en clases más generales. Por ejemplo, tal vez no se pueda identificar exactamente que es una llamada por Skype, pero sí se puede saber que se está enviando un codec de VoIP porque se observa una cadencia fija y eso sea suficiente para hacer una clasificación del servicio. Algunas de las desventajas que ofrece este método de clasificación son:

- **No es dinámico:** En una red real es complicado monitorizar todas las aplicaciones populares que utilizan los usuarios. Además, los ciclos de popularidad de las aplicaciones cada vez son más cortos. Esto puede hacer que las características de los flujos de red varíen drásticamente en poco tiempo. [27] Por ejemplo, aplicaciones como TikTok o Clubhouse, presentan nuevos medios de comunicación y difusión audiovisual, completamente distintos a un sistema clásico como Youtube o Spotify.

- **Introduce latencia:** Para realizar correctamente los cálculos asociados los estadísticos de un flujo, se requiere de la llegada de varios paquetes del mismo. Esto implica que el flujo de datos quedará paralizado hasta que se reciban y con ello se introduce una latencia que, en muchos casos, no es asumible en una red real. Esto solo ocurrirá en el caso de que se haga a tiempo real, si se hace a posteriori o no se clasifican los primeros paquetes no afectará. Por ejemplo, se puede tener una clase por defecto que no tenga prioridad.
- **Costes computacionales y de memoria:** Como se ha explicado en el apartado anterior, se requiere hacer cálculos sobre varios paquetes por flujo. Esto implica que se deben almacenar en alguna memoria de tipo RAM. Sin embargo, en una red de solo 100 Mbit/s se pueden estar generando en el peor caso entre 1 y 5 millones de flujos de tráfico. [28] Encontrar una memoria RAM capaz de manejar tal cantidad de flujos en redes de 100 Gbps resulta imposible. Se debe buscar algún tipo de procesamiento al vuelo.

Por último, otro sistema que puede utilizarse es la correlación de distintos flujos de tráfico. Por ejemplo, si el gestor de red tiene acceso al DNS, podría saber que tipos de aplicaciones está solicitando cada usuario. Sin embargo, no es tan fácil tener acceso a dicho DNS, muchas conexiones no necesitarán de solicitud al DNS y este sistema queda obsoleto con el uso del DoH (*DNS over HTTPS*, DNS sobre HTTPs), pues se transmitiría esta información cifrada.

2.2.4. Clasificación de tráfico cifrado

Como se ha explicado en los apartados anteriores, la clasificación de tráfico cifrado resulta un gran desafío, debido principalmente a que no se puede acceder al contenido de los datos, y las redes reales tienen requerimientos muy estrictos de rendimiento en latencia y tasa.

Sin embargo, en la actualidad, varios autores han establecido una metodología de clasificación de tráfico cifrado que resulta prometedora mediante el uso de aprendizaje automático para identificar a qué clase pertenecen paquetes individuales. En la **Tabla 2.3** se describen algunos de los artículos leídos.

2.3. Aprendizaje Automático

El aprendizaje automático (o *machine learning*) es un subcampo de las ramas de la computación que desarrolla técnicas para que los sistemas puedan aprender a realizar tareas a partir de los datos, sin ser explícitamente programados mediante algoritmos específicos. Como se observa en la **Figura 2.4**, el aprendizaje automático se diferencia de la computación clásica en que aprende de los datos en vez de un algoritmo dado, en ese aspecto intenta imitar el aprendizaje humano.

El aprendizaje automático tiene numerosas ventajas frente a la programación clásica. Permite desarrollar sistemas para realizar tareas específicas sin la necesidad de tener conocimientos específicos y profundos sobre el problema a resolver (eso no quiere decir que

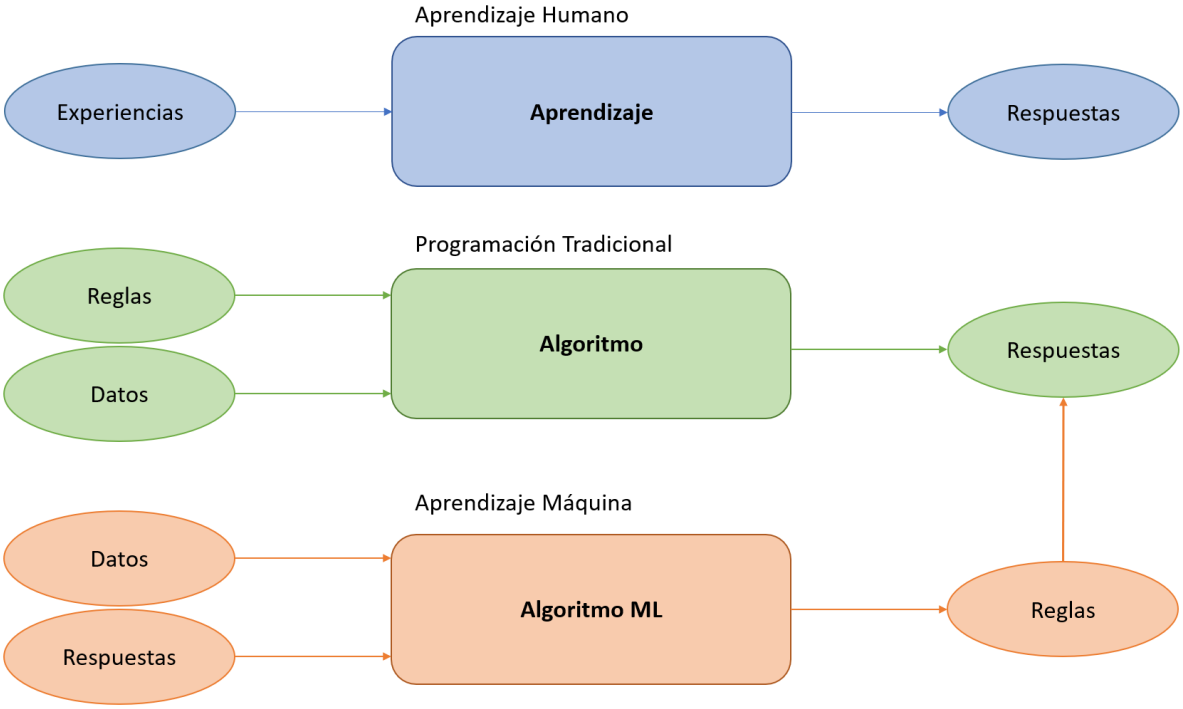

Figura 2.4: Diferencia entre Aprendizaje Humano vs. Aprendizaje Máquina.

no ayude el tener esos conocimientos, pues facilita la interpretabilidad de los resultados). También permite usar datos no estructurados para resolver problemas, mediante modelos de redes neuronales o algoritmos de *clustering*. Además, muchos algoritmos de aprendizaje automático están basados o imitan el aprendizaje humano.

Hay varios tipos de algoritmos de aprendizaje automático, como pueden ser los algoritmos de reducción y aumento de dimensionalidad, los algoritmos de agrupamiento, máquinas de soporte vectorial, los algoritmos de regresión, los árboles de decisión o los algoritmos bayesianos. [29] Sin embargo, en este Trabajo Fin de Máster no vamos a entrar en detalle en cada uno de ellos. Nos centraremos en los algoritmos de redes neuronales y dentro de ellos, los de aprendizaje profundo. Los algoritmos de aprendizaje automático anteriores en general se agrupan según su funcionamiento. Estos son:

- **Aprendizaje supervisado:** Este sistema de aprendizaje etiqueta los datos previamente. El algoritmo aprenderá en función de la capacidad que tenga de predecir la etiqueta para un conjunto de datos nuevos. En este tipo de algoritmos encontramos los de regresión y clasificación. [30]

- **Aprendizaje no supervisado:** Este tipo de aprendizaje se utiliza en aplicaciones de *clustering*. Se parte de un conjunto de datos no etiquetados y el algoritmo deberá encontrar patrones y agrupar los datos parecidos.

- **Aprendizaje por refuerzo:** Este tipo de aprendizaje está basado en la psicología conductista, en la cual, un proceso deseado será premiado mientras que un proceso no deseado será castigado. De esta forma con suficiente entrenamiento se logrará obtener un aprendizaje. [31]

2.3.1. Conceptos fundamentales del funcionamiento de las redes neuronales

En este apartado se hará una revisión de los conceptos fundamentales necesarios para entender el funcionamiento de las redes neuronales.

- **Neurona artificial:** En una red neuronal, el elemento mínimo de procesamiento sería una neurona. Al igual que un cerebro biológico, las neuronas son estructuras interconectadas entre ellas cuya función es transmitir la información. En el caso de las neuronas artificiales, se conectarán a otras neuronas y según corresponda ponderarán el peso que tiene dicha conexión. Como se observa en la **Figura 2.5** una neurona tendrá entradas (*inputs*) y salidas (conexiones o *outputs*), estas conexiones a otras neuronas serán permiten la propagación de la información y se agrupan en las capas de la red neuronal.

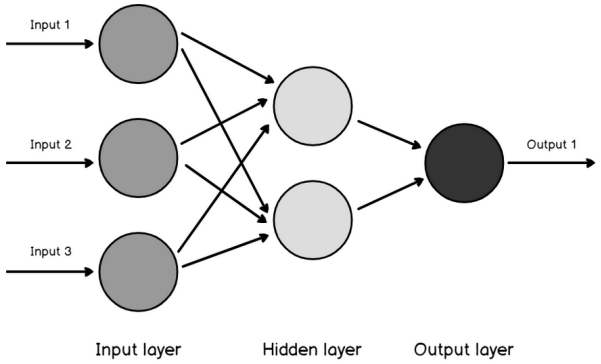

Figura 2.5: Diagrama de conexiones de tres capas densamente conectadas. [4]

- **Capas:** Las capas de una red neuronal se componen de múltiples neuronas interconectadas de alguna manera (más adelante se verán qué formas se pueden utilizar). Las distintas distribuciones de neuronas en una capa darán propiedades especiales a cómo procesa y distribuye la información. Es por eso que, en una red neuronal, se debe buscar conectar las distintas capas de forma inteligente, de manera que las propiedades individuales de cada capa ayuden a resolver una parte del problema planteado. A este conjunto de capas se le puede llamar modelo. En la **Figura 2.6** se muestra el diagrama de una red neuronal formada por capas convolucionales.

- **Conjunto de datos:** En una red neuronal, se necesitan datos para poder entrenar el modelo para resolver una tarea concreta. Sin embargo, cuanto más compleja sea una red neuronal, mayor cantidad de datos requerirá para poder ser entrenada. Es por eso que se debe ser inteligente a la hora de dividir un conjunto de datos finito, pues hacen falta datos para entrenar, testear y validar.

  Los datos de entrenamiento servirán para determinar los pesos de las conexiones de las neuronas en el proceso del entrenamiento. Los datos de testeo permitirán comprobar los resultados de aciertos que se están obteniendo en cada fase del desarrollo. Por último, se reservan un conjunto de datos para la validación final del modelo.

  En el desarrollo de este Trabajo Fin de Máster no se propone un despliegue en entorno real del modelo. Por lo tanto, se pueden usar los datos de validación como

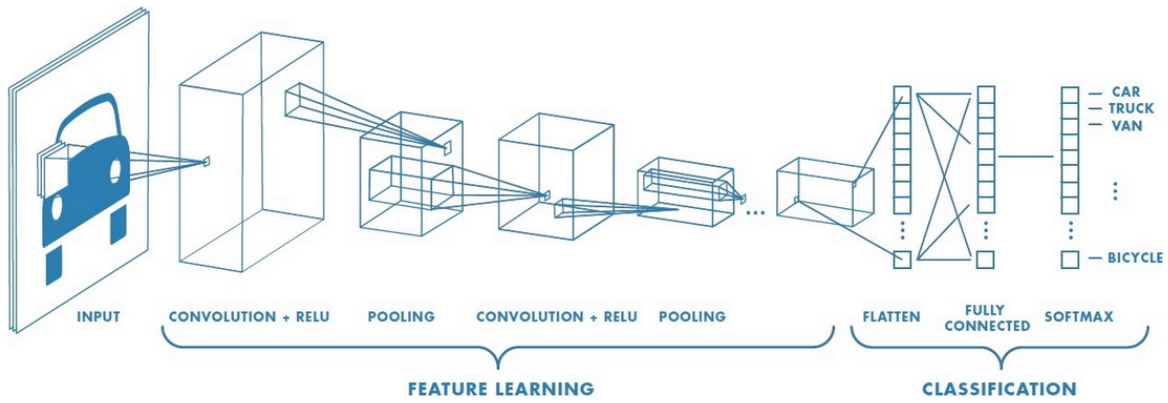

Figura 2.6: Red neuronal formada por capas convolucionales. [5]

datos en el conjunto de testeo. El reparto de los datos entre entrenamiento y testeo debería estar entre el 60 % - 40 % y el 80 % - 20 %. [32] En la **Figura 2.7** se representan un esquema de la división de un conjunto de datos. Para este proyecto se ha elegido una distribución de 70 % datos de entrenamiento y 30 % datos de testeo, ya que suele ser una división bastante común en el campo del *Deep Learning*[33].

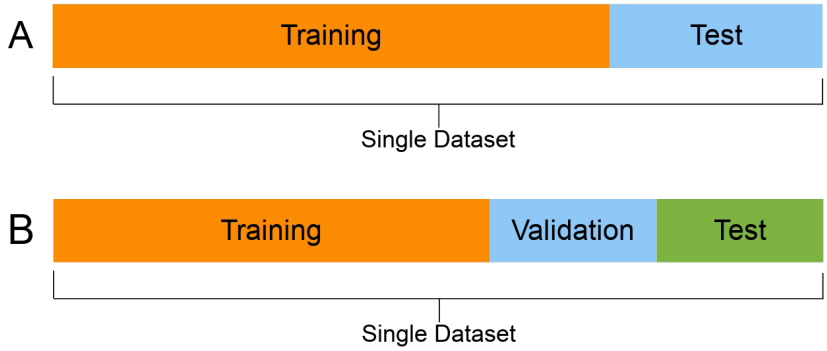

Figura 2.7: División de un conjunto de datos. [6]

- **Tamaño de Lote:** El entrenamiento de una red neuronal puede ser un proceso muy costoso computacionalmente, por ello, se suelen utilizar sistemas CPU + GPU que permiten paralelizar las operaciones, consiguiendo velocidades de entrenamiento varios órdenes superiores con respecto a un sistema CPU sin paralelismo en los procesos. Para ello, se debe definir un valor de tamaño de lote de datos, que serán el número de datos que se utilizarán. Además, en un despliegue de la aplicación, este valor será el que determine cuantos datos se procesan a la vez.
- **Época o iteración:** En las redes neuronales el entrenamiento se realiza según iteraciones, en cada iteración se procesan los datos de entrenamiento para calcular los pesos de las conexiones de las neuronas. Al comienzo del entrenamiento se parte de un proceso semi-aleatorio que podría dar resultados sub-óptimos. Por ello, se debe entrenar la red neuronal varias veces, para asegurarse de que está encontrando la solución al problema con los datos dados.
- **Función de coste:** La función de coste es la herramienta que permite a la red entrenar. Partiendo de los datos dados y una etiqueta concreta se calcularán los

valores de los pesos de las neuronas que minimicen la diferencia entre la entrada y la salida para una función de coste dada.

- **Tasa de aprendizaje:** Para una función de coste dada y un número de iteraciones se puede determinar si la red neuronal está aprendiendo correctamente. Para ello, se utilizan los datos de testeo que valorarán si para una etiqueta dada se está acertando o no. Así se puede ver como mejora o empeora en cada época del entrenamiento la precisión de la red. La tasa de aprendizaje determinará la velocidad a la que aprende dicha red, sin embargo, no se deben usar valores muy altos pues podrían desestabilizar el proceso de entrenamiento. Tampoco se deben utilizar valores muy bajo, porque entonces las diferencias en los pesos de las conexiones de neuronas, entre cada iteración serán muy bajos y podría no aprender la red debido a que se optimiza la función de coste en mínimos locales. También existe el problema del sobre-ajuste, que se comenta en el siguiente apartado.

  El concepto opuesto a la tasa de aprendizaje serían las pérdidas, que contabilizan qué tan incorrecta fue la predicción del modelo. De esta forma, se puede medir si un modelo no ha entrenado correctamente tendrá pérdidas altas, cuando se vaya ajustando el modelo las pérdidas se irán reduciendo.[34]

  En la **Figura 2.8** se muestran tres ejemplos de distintas tasas de entrenamiento. En la gráfica (A), se observa el caso de sobre-ajuste e infra-ajuste. En el caso (B), se muestra una tasa de entrenamiento adecuada, una alta y una lenta. Por último, en el caso (C) aparece un entrenamiento convergente y uno no convergente.

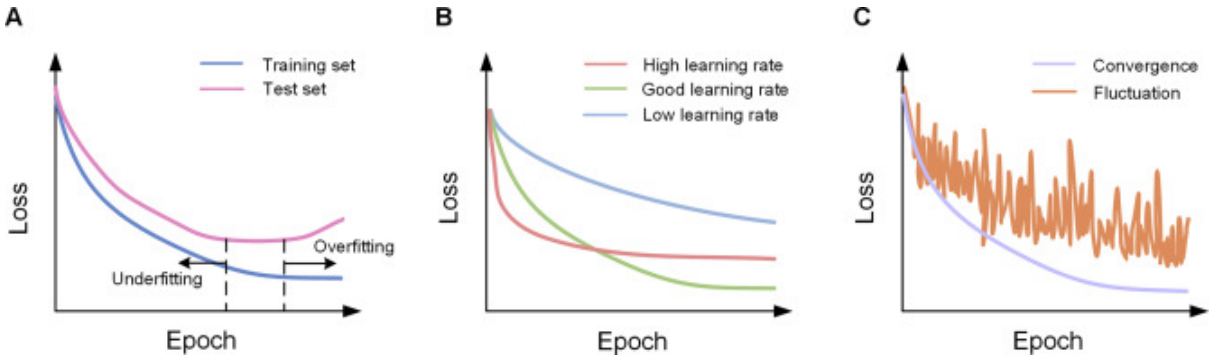

Figura 2.8: Comparación de tasa de aprendizaje. [7]

- **Problema del sobre-ajuste:** Como se ha visto en la imagen anterior (**Figura 2.8**), en el proceso de entrenamiento se puede dar un problema denominado sobre-ajuste (*overfitting*). Este ocurre cuando la red tiene precisión mucho mayor en los datos de entrenamiento que en los de testeo. Esto quiere decir que la red no está aprendiendo, sino que está copiando. Es muy importante conseguir que la red aprenda y sea capaz de inferir la información de los datos para poder resolver el problema en un contexto general. Las soluciones al *overfitting* son varias: aumentar los datos de entrenamiento; balancear los datos entre las distintas clases; usar capas de *dropout*; usar regularización del *kernel* o reducir la complejidad el modelo.

  Otro problema similar es el sub-ajuste o *underfitting*, en este caso el modelo no es suficientemente complejo como para solucionar el problema dado. Un ejemplo típico sería: hacer una red neuronal que clasifique razas de perros, pero en el entrenamiento

solo usamos perros con piel oscura. Al testear el modelo con un perro blanco, como un bichón maltés, el modelo no será capaz de reconocerlo correctamente.

En los siguientes apartados se expondrán las redes neuronales más típicas para resolver el problema de la clasificación de tráfico cifrado.

2.3.2. Redes neuronales convolucionales

Las redes neuronales convolucionales son aquellas que utilizan capas convolucionales. Esto quiere decir que las operaciones básicas de sus capas son la convolución. Por ese motivo estas redes trabajan con las estructuras espaciales de los datos y por eso pueden ser muy útiles para analizar y clasificar imágenes.

En la **Figura 2.9** hay un ejemplo típico de red convolucional, AlexNet, que utiliza convoluciones bidimensionales en imágenes. Esto permite a la red procesar la información desde capas más generales a capas más específicas. Las redes neuronales convolucionales utilizan tres tipos de capas: capas convolucionales, capas de agrupación y capas totalmente conectadas. (*convolutional layers, pooling layers y fully-connected layers*). Las capas convolucionales realizan operaciones de convolución sobre imágenes en 2D o 3D. Esto permite extraer las características de la imagen. Las capas de agrupación se encargan de reducir la dimensión espacial de la imagen. Para ello, se aplica la operación de *MaxPooling*, que obtiene el valor máximo de los píxeles filtrados. En un principio, este proceso puede aparentar que produce una pérdida de información en la red. Sin embargo, resulta beneficioso pues permite obtener características más específicas de la imagen. Por último, están las capas totalmente conectadas, que también se llaman densamente conectadas. Su operación más típica es conectar todas las neuronas de una capa con todas las neuronas de la capa siguiente, se suelen utilizar en las capas finales y tienen el objetivo de procesar las características aprendidas en las capas anteriores y traducirlas en alguna de las clases de la clasificación.

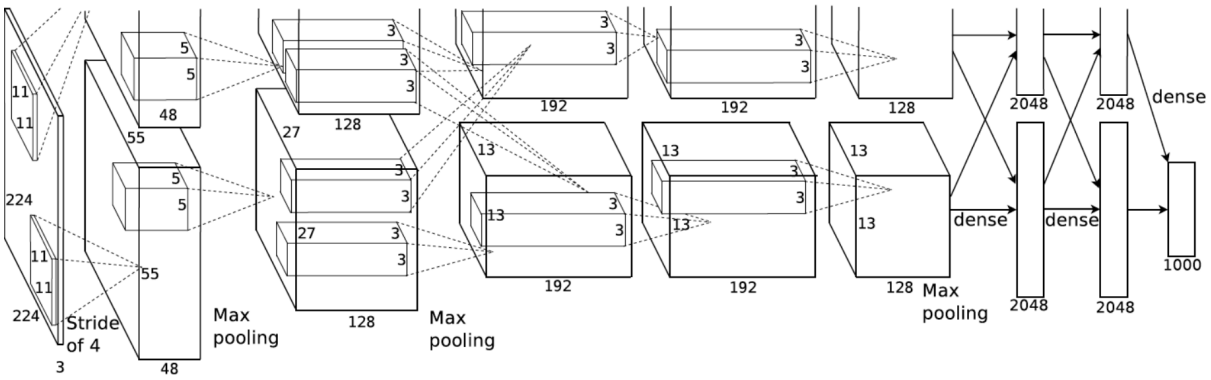

Figura 2.9: Modelo Alexnet para clasificación de imágenes. [8]

2.3.3. Redes neuronales recurrentes

Las redes neuronales recurrentes son aquellas que incluyen bucles de retroalimentación entre las capas. Estos bucles permiten que información de capas posteriores pueda ser

recibida como datos en las capas anteriores. Esto permite a la red neuronal trabajar con estructuras temporales en los datos. Por eso, son muy utilizadas en analizar y clasificar audio o texto. Cuando son usadas junto con redes convolucionales pueden servir para procesar vídeos.

Uno de los tipos de redes recurrentes más utilizados son las LSTM (*Long Short Term Memory*, Redes de Memoria a Largo y Corto plazo). Estas neuronas tienen bucles de entrada y salida que interactúan con una pequeña celda de memoria. Al conectar varias celdas, la red empieza a adquirir propiedades de escritura, actualización y borrado de datos, como si de una memoria artificial se tratase. Estos modelos de red recurrente son muy utilizados en sistemas de modelado del lenguaje, reconocimiento del habla o traducción automática. En la **Figura 2.10** hay una comparación entre los distintos tipos de neuronas recurrentes.

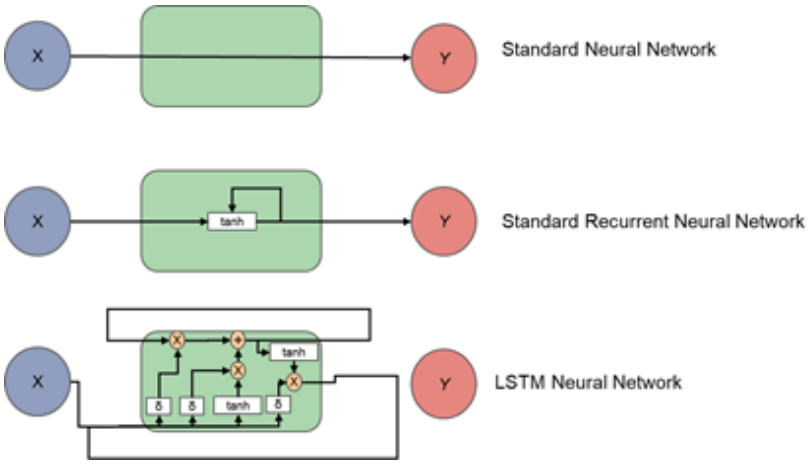

Figura 2.10: Comparación de neuronas recurrentes y LSTM. [9]

2.4. Clasificación de tráfico con redes neuronales

Según lo visto hasta el momento, cabe plantearse cómo podría ayudarnos el aprendizaje automático y las redes neuronales a solucionar el problema de la calificación de tráfico cifrado. En uno de los primeros artículos que se publicaron sobre utilizar aprendizaje profundo para resolver un problema de clasificación de tráfico [13] se expone como detectar trafico malicioso mediante el uso de redes convolucionales, que en esencia es un problema parecido al de clasificación de aplicaciones.

Para ello, los autores procesan los paquetes del tráfico de red para mostrarlo como si fuesen imágenes 2D (en escala de grises) y se observa que, dependiendo del tipo de aplicación que sea, pueden ver patrones específicos.[35] Esto es muy interesante, porque significa que aunque el tráfico este cifrado hay una forma de clasificarlos según la aplicación de origen. De esta forma, un ISP podría ejecutar sus políticas de diferenciación de servicio a la vez que los datos de los usuarios quedan protegidos. Para poder explotar estos patrones únicos se recurre a un modelo de clasificación de imágenes mediante redes neuronales convolucionales. Estas han sido muy investigadas en los últimos años para resolver problemas de clasificación de imágenes reales con éxito.

Distintos autores han aplicado distintas técnicas relacionadas con el aprendizaje automático en clasificación de tráfico. Se han utilizado distintos *datasets*, modelos, aplicaciones y metodologías como Perceptrón Multicapa, CNN, RNN, *Autoencoders y fullyconected layers*. En la actualidad este método se ha asentado como una de las principales líneas de investigación en el área de la clasificación de tráfico. Sin embargo, pocos estudios han planteado cómo clasificar en una red real y los que lo han planteado ha sido con dispositivos GPU, los resultados son de 0.18 ms por paquetes, pero utilizan lotes de 64 imágenes. [36] En la lectura de artículos y estudios sobre el tema que se ha realizado en este Trabajo Fin de Máster, se han detectado algunas mejoras posibles. En la **Tabla 2.2** y en la **Tabla 2.3** se muestra un resumen de los distintos trabajos realizados en campo de la clasificación de tráfico cifrado.

Tabla 2.2: Resumen de los sistemas de clasificación propuestos I. [17]

Work	DL Algorithms	Model Input	Dataset	Classification Task	Granularity	Year
Z.Wang [19]	MLP,SAE	raw packet	private	traffic classification	application	2015
Datanet [20]	MLP,SAE,CNN	raw packet	ISCX2012	encrypted traffic classification	application	2018
DeepPacket [21]	MLP,SAE,CNN	raw packet	ISCX2012	encrypted traffic classification	application	2018
Wang-2D-CNN [22]	CNN	raw data	USTC-TFC2016	malware classification	application	2017
Wang-1D-CNN [23]	CNN	raw data	ISCX2012		encrypted traffic classification	2017
Lopez-Martin [24]	CNN,LSTM	packet-level features	private dataset	traffic classification	application	2017
Aceto [25]	MLP,SAE,CNN,LSTM	raw data	private dataset	mobile traffic classification	application	2018
HAST-IDS [26]	CNN,LSTM	raw data	ISCX2012	intrusion detection	application	2018
Rezaei [27]	CNN	packet-level features	private dataset	QUIC encrypted traffic classification	application	2018
Van [28]	CNN,RF	packet-level features	private dataset	QUIC encrypted traffic classification	application	2018
Y.Wang [29]	CNN	flow features	Moore	traffic classification	services group	2017
Seq2Img [30]	CNN	raw data	private dataset	traffic classification	protocol/application	2017
Li [31]	MLP,VAE	raw data	IMTD17	mobile traffic classification	application	2017
Vu [32]	GAN	raw data	NIMS	encrypted traffic classification	protocol	2017

Se detecta otro problema consistente en que varios estudios utilizan en la red neuronal las características de flujo en vez de imágenes de los paquetes. Esto en sí no es un error

Tabla 2.3: Resumen de los sistemas de clasificación propuestos II.

Work	Algorithm	Model Input	Dataset	Classification Task	Granularity	Year
R.Hwang [19]	CNN + RNN	Packet-level features	ISSCX2012 USTC-TFC2016, IoT	IDS malicious traffic	Application	2019
X.Zeng [35]	CNN	Raw-traffic	USTF-TFC2016	Malware traffic	Application	2017
A. Jurcut [37]	LSTM	Packet-level features	NSL-KDD	IDS for U2R & R2L	Attack Category	2020
S. Razaei [38]	CNN + LSTM	Adjacent Flows	"Large mobile traffic dataset"	Mobile class application traffic	Application	2017
V. Tong [39]	CNN	Flows & packet-level features	"Network flow of quic dataset"	QUIC-based services	Application	2018
Y. Zeng [?]	CNN + LSTM & SAE	Packet-level features	ISCX VPN non VPN & ISCX 2012 IDS	Traffic classification & IDS	Application	2019
M. Lotfolla [13]	CNN + LSTM & FC	Packet-level features	UNB ISCX VPN non VPN	Traffic Characterization & Application Identification	Application	2020
A. Mahadi [40]	ANT	Flows & packet-level features	ISCXVPN2016	Traffic Characterization & Application Identification	Application	2019

metodológico, de hecho es un mecanismo muy efectivo para la clasificación de tráfico. Sin embargo, cae en el mismo error que se comentó en el apartado 2.2.2, en una red real actual se deben tener en cuenta la latencia introducida por el sistema de captura de flujos, así como los costes computacionales y de memoria que genera dicho proceso. La solución a este problema es procesar los paquetes al vuelo, sin esperar a tener varios paquetes de un mismo flujo.

El último problema detectado serían que los distintos artículos no dejan claro cómo dividen los paquetes de entrenamiento y test. En el caso de tráfico de red, es especialmente importante dividirlos correctamente, para evitar el sobre-ajuste. Por ejemplo, si usamos una captura de tráfico de 100 paquetes y cogemos los primeros 70 para entrenamiento y los últimos 30 para test. Estaremos entrenando el modelo para reconocer la fase de conexión pero no la de cierre, que se corresponde con los últimos paquetes. De esta forma, el modelo tendrá una precisión limitada en un entorno real. Pero entonces, ¿queda resuelto este problema si se toman los datos de entrenamiento y test en orden aleatorio?. Pues tampoco tiene por qué ser así, al usar paquetes aleatorios probablemente se estén utilizando paquetes del mismo flujo o sesión en el entrenamiento y el test. Esto no solo no es un caso realista, sino que además está propiciando que la red aprenda copiando valores que definen un flujo (como sus direcciones IPs o puertos) frente al contenido de los datos. Incluso si se descartan las direcciones IPs y los puertos es probable que haya información en los datos que sea única del flujo, como identificadores de sesión o claves de cifrado que nunca se van a repetir en una red real, pero que el modelo aprenderá a utilizar cómo patrones de una aplicación.

2.5. El problema de la clasificación de tráfico a tiempo real en redes avanzadas

Si bien las redes neuronales han demostrado ser capaces de obtener buenas medidas de precisión al clasificar varias clases de tráfico cifrado, queda por resolver el problema de las prestaciones necesarias para analizar el tráfico en entornos reales, con baja latencia y alta tasa de procesamiento.[36] Los sistemas de CPU + GPU pueden trabajar a altas tasas de procesamiento, pero resultan ineficaces en cuanto a latencia. Esto se debe a que dichos sistemas requieren de interacción con memorias y llamadas a funciones desde sistemas operativos. Ambos procesos pueden ser lentos cuando desea minimizar la latencia. En algunos artículos, los sistemas de clasificación con GPUs obtienen rendimientos de hasta 35 Gbit/s y 4 o 5 ms de latencia.[41]. En otros artículos llegan a tiempos de detecciones de milisegundos utilizando una tarjeta gráfica Nvidia Tesla P100.[42]

En la actualidad, los sistemas empotrados han evolucionado enormemente, y se posicionan como alternativas a los PCs clásicos para resolver diferentes tareas específicas. En concreto, los sistemas basados en FPGA cumplen con los requisitos que se buscan para resolver la tarea propuesta de minimizar la latencia de procesamiento.

2.6. Lógica Programable (FPGAs)

Xilinx, la mayor empresa fabricante de FPGAs las define como: dispositivos semiconductores basados en una matriz de bloques lógicos configurables conectados mediante interconexiones programables. Esto quiere decir que son dispositivos con la capacidad de reconfigurar las conexiones entre transistores para poder programar funcionalidades o aplicaciones específicas en el *hardware*. Esto permite a las FPGAs ser programadas para realizar aplicaciones específicas con prestaciones muy superiores a un sistema de procesadores de uso general. En la actualidad, los fabricantes de FPGAs están pivotando a unidades SoC (*System-On-Chip*, Sistema en Chip), es decir, la FPGA es solo una parte de la placa y esta incluye sistemas de memoria, IO (*Input-Output, Entrada-Salida.*), GPU o aceleradores de AI (*Artificial Intelligence*, Inteligencia Artifical). La **Figura 2.11** muestra un dispositivo Xilinx Zynq Ultrascale+ ZCU104, que es el que se utilizará en este proyecto.

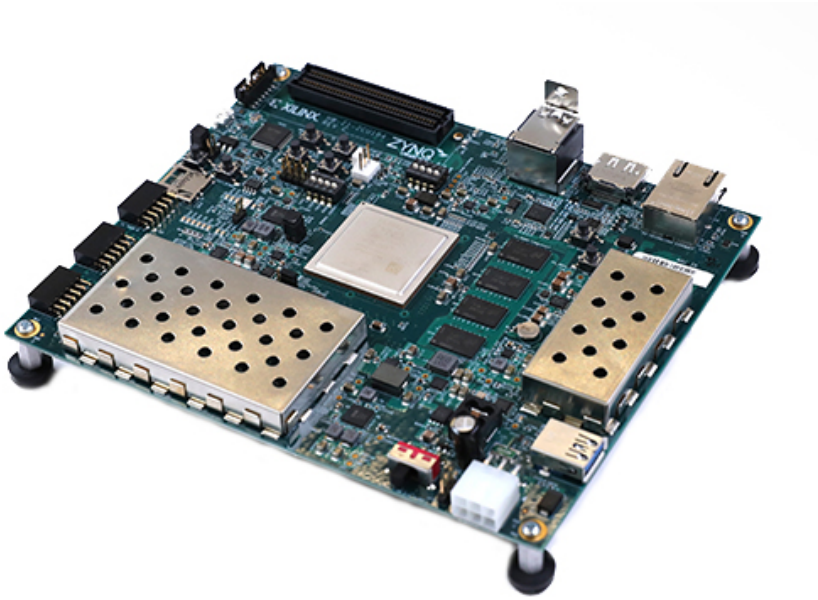

Figura 2.11: Fotografía del dispositivo FPGA ZCU104. [10]

2.6.1. Flujo de diseño FPGA

El flujo de diseño tradicional en FPGA parte de descripciones RTL (*Register Transfer Level*, Lenguaje de Transferencia de Registros) en HDL (*Hardware Description Language*, Lenguajes de Descripción de Hardware) como Verilog o VHDL. En la última década se ha tendido a subir el nivel de abstracción, por ejemplo, usando descripciones desde C/C++ con la síntesis de alto nivel (HLS - *High Level Syntesis*), desde Matlab usando Simulink y la integración de *cores IP* con la herramienta *IP Integrator*.

Así mismo, se han desarrollado herramientas específicas para ciertos dominios de aplicación, como aceleración de algoritmos (SDAccel), desarrollo de empotrados (SDSoC), o aplicaciones de red (SDNet). Dentro de estas herramientas aplicadas a dominios de aplicación, Xilinx ha desarrollado herramientas para portar a sus dispositivos modelos

de ML/AI. Al margen de otras herramientas que existen y han existido en esta área, el nombre comercial de ese conjunto de herramientas recibe el nombre de Vitis-AI.

2.6.2. Framework para ML/AI: Xilinx Vitis AI

Aun con las herramientas de alto nivel actuales, para poder acelerar una aplicación usando FPGAs se deben seguir unos pasos de diseño concretos. Primero, se desarrolla el sistema en lenguaje C o C++, y tras el proceso de síntesis de alto nivel y la integración de diferentes *IP-cores*. Luego el proceso de síntesis RTL, emplazado y rutado se culminan con la generación del *bitstream* que permite programar la FPGA.

Esto no es problema en la mayoría de aplicaciones. Sin embargo, el lenguaje más usado para aplicaciones de aprendizaje automático es Python. Gracias a las bibliotecas de Google de Tensorflow y Keras, los ciclos de desarrollo de aplicaciones son mucho más rápidos que en otros entornos como Caffe o Pytorch. Además, la gran comunidad de usuarios y recursos disponibles que hay permiten realizar prototipos muy rápidamente. Por ese motivo, Xilinx desarrolló el entorno de trabajo Vitis AI, que funciona con Tensorflow, Caffe y Pytorch. La **Figura 2.12** muestra un esquema del entorno de desarrollo Xilinx Vitis AI.

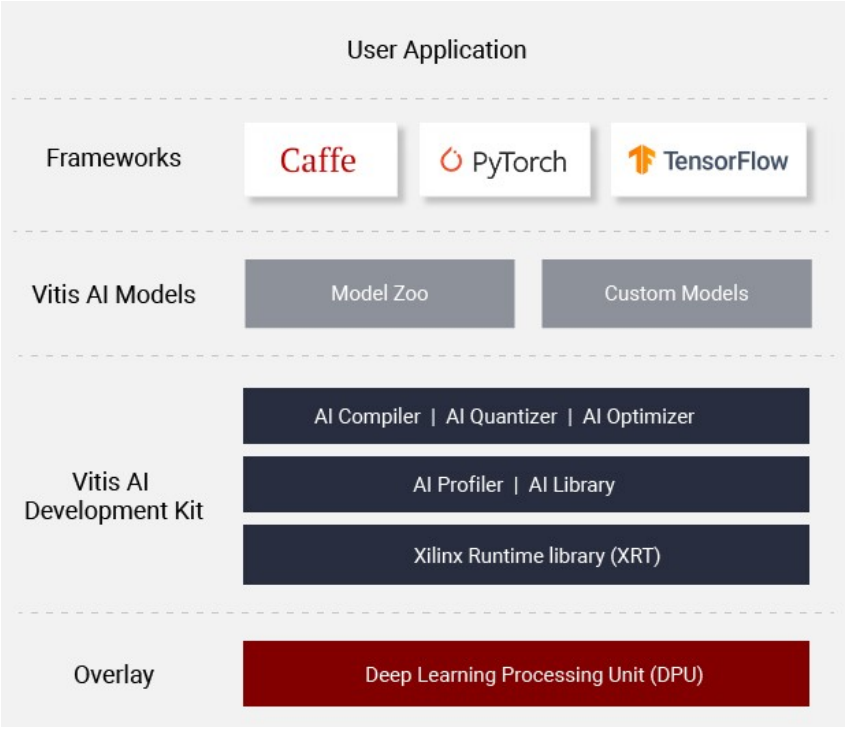

Figura 2.12: Flujo de trabajo en Xilinx Vitis AI. [11]

El entorno de desarrollo permite realizar prototipos de modelos de redes neuronales y desplegarlos en plataformas hardware como FPGAs de tipo Alveo o los equipos Zynq Ultrascale+. El *stack* de desarrollo Vitis AI contiene los siguientes componentes: [11]

- **AI Model Zoo**: Para realizar prototipos rápidos de aplicaciones y comprobar que la FPGA presenta mejores valores que una CPU. Xilinx añade un conjunto de modelos pre-optimizados que están listos para ser desplegados en dispositivos Alveo o Zynq.

- **Cuantificador AI**: El cuantificador permite optimizar el tamaño de nuestra AI hasta un 90 %. Sirve para reducir el número de conexiones que tienen las neuronas, así como la resolución de los pesos de estas. Este proceso de calibración se realiza para asegurar que la red cumple con requisitos de alta tasa y baja latencia, a la par que reduce el consumo de memoria y mejora la eficiencia energética. Es normal que en este proceso se pierda algo de precisión en el modelo, pero hay un proceso opcional, que es el Optimizador, que permite reentrenar un modelo cuantizado para maximizar la precisión del modelo.

- **Compilador de IA**: El modelo cuantificado se debe convertir a un formato que pueda entender la FPGA. En el proceso de compilado se compila el modelo cuantificado para obtener un conjunto de instrucciones y un flujo de datos de alta eficiencia. En la **Figura 2.13** se representa proceso de AI Quantizer y AI Compiler, se observa que de una red neuronal de 32 bits se pasa a una de 8 bits y luego se generan las instrucciones binarias para la DPU.

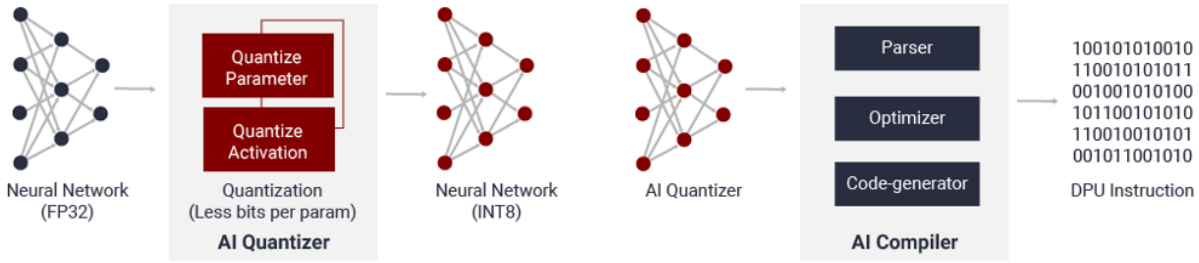

Figura 2.13: Proceso de AI Quantizer y AI Compiler. [12]

- **AI Profiler**: Realiza un análisis en profundidad de la eficiencia y utilización de la implementación de la inferencia de la IA.

- **Biblioteca de IA**: Ofrece APIs de C++ de alto nivel pre-optimizadas para aplicaciones de IA desde el borde de la red hasta la nube.

- **DPU**: Núcleos IP eficientes y escalables que se pueden personalizar para satisfacer las necesidades de muchas aplicaciones ML diferentes.

Las FPGAs se han convertido en dispositivos muy utilizados para aplicaciones de IA en "computación en el borde". Por ejemplo, los vehículos autónomos, ciudades inteligentes, o industria inteligente.

2.6.3. Python en FPGA: Pynq

Pynq es un proyecto de código abierto de Xilinx que facilita el diseño de sistemas integrados de la plataforma Zynq de Xilinx. Usando el lenguaje Python y sus bibliotecas, los diseñadores pueden aprovechar los beneficios de las FPGAs y los microprocesadores presentes en sus SoC (*System-on-Chip*, Sistemas en un Chip). El objetivo es que los usuarios puedan crear aplicaciones de alto rendimiento con las ventajas que ofrecen las FPGAs como son: ejecución paralela en hardware, procesamiento de vídeo de alta tasa,

aceleración por hardware de algoritmos, procesamiento de señal a tiempo real, IO de alto ancho de banda o baja latencia. [43]

En concreto, Pynq Permite a los arquitectos, ingenieros y programadores diseñar en sistemas embebidos sin tener que utilizar las herramientas de diseño de tipo ASIC para circuitos programables. Esto lo consigue de tres maneras diferentes: [44]

- ***Overlays:*** Son como las bibliotecas de *software* pero para los circuitos programables en *hardware*. De esta forma se podrá acceder a las instrucciones del sistema *hardware* mediante una API (*Application Programming Interface*, Interfaz de Programación de Aplicaciones).
- **Python:** PYNQ utiliza el lenguaje Python para programar los procesadores embebidos. En verdad, utiliza CPython que está escrito en C, e integra miles de librerías de C. Además permite usar instrucciones básicas en C y C++ de bajo nivel para obtener un sistema eficiente.
- **Código abierto:** PYNQ es un proyecto de código abierto que adopta una arquitectura basada en web. Para interactuar con el *kernel* Python se incorpora un *Jupyter Notebook*, de esta forma, se podrá utilizar cualquier buscador web para desarrollar en la plataforma.

En este trabajo se utiliza la infraestructura Pynq para poder simplificar la interacción con los aceleradores hardware de Inteligencia Artificial (DPU) desde un entorno Jupyter. El *overlay* de Pynq DPU está diseñado para VitisAI e incluye los *bitstreams* necesarios para poder ejecutar modelos en una FPGA de tipo Zynq Ultrascale+. Este sistema nos permite tener una imagen con una DPU funcional en la que podemos ejecutar nuestra red neuronal de clasificación de tráfico. En la **Figura 2.14** se representa un esquema con los componentes de la Pynq DPU.[20]

Con el sistema Pynq DPU se podrá ejecutar el modelo de la red neuronal en FPGA de una forma más sencilla, evitando tener que realizar el proceso de diseño hardware y abstrayendo la implementación de la FPGA a un desarrollo de Python en una interfaz web. Esto resultará beneficioso para prototipar el clasificador de tráfico y realizar mediciones para comparar con el sistema de procesamiento CPU + GPU.

2.7. Conclusión

En este capítulo se han explicado las bases necesarias para poner en contexto el diseño y desarrollo del clasificador de tráfico. En concreto, se ha puesto en contexto como se clasifica tráfico y las dificultades que existen dependiendo del sistema. También se ha expuesto como los nuevos modelos de redes neuronales son aparentemente capaces de resolver la clasificación de tráfico cifrado.

Por último, se ha explicado cuál es el funcionamiento de las FPGAs, las nuevas arquitecturas DPU y como pueden ayudar a resolver los problemas derivados de la clasificación de tráfico de red a con baja latencia y sin disminuir la calidad de experiencia de los usuarios.

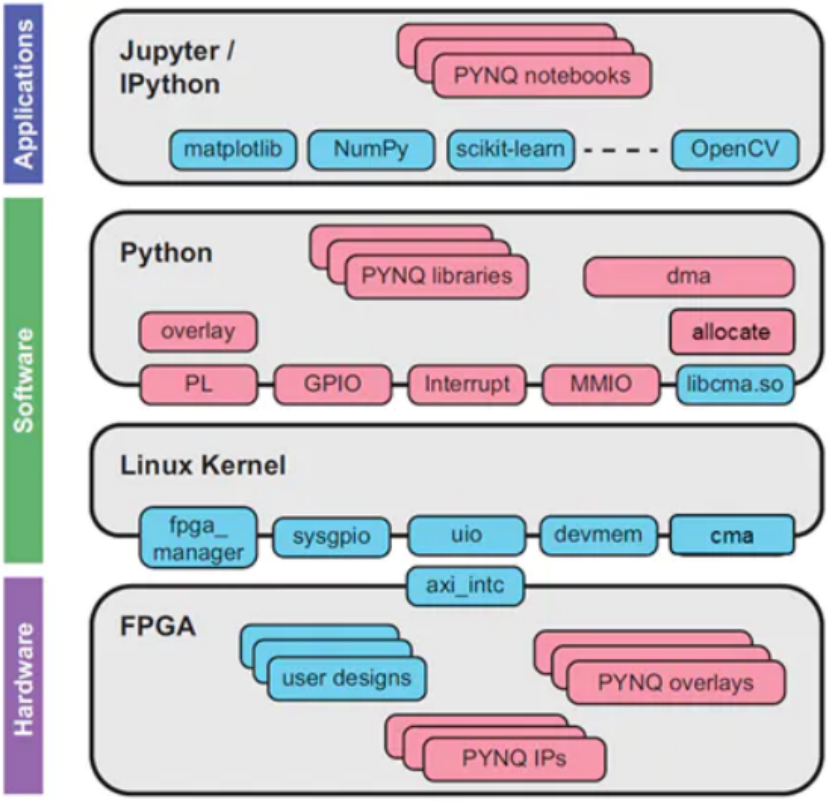

Figura 2.14: Entorno de trabajo Pynq DPU.

Una vez se ha estudiado el estado del arte se tienen los conocimientos necesarios para abordar el proyecto. Se debe comenzar planteando los requisitos que debe cumplir el clasificador de tráfico, luego se propondrá un diseño y se documentará el desarrollo.

3

Análisis de Requisitos

3.1. Introducción

En este capítulo se procede a analizar los requisitos que debe cumplir el clasificador de tráfico para llevar a cabo su función.

Los requisitos funcionales serán aquellos que resultan imprescindibles para el funcionamiento del disector. Sin ellos, el comportamiento del sistema será erróneo o deficiente. Los requisitos no funcionales, serán aquellos que aumenten la facilidad de usar el sistema y permitan al usuario tener una experiencia cómoda usando el programa. También se reserva un apartado para comentar el entorno de trabajo en el que se va a desarrollar este Trabajo Fin de Máster.

3.2. Descripción del problema

Imaginemos que un gestor de red desea saber qué aplicaciones están utilizando los usuarios, tal vez por motivos de seguridad o de optimización de los recursos. Para ello, debe clasificar el tráfico de red, pero debido a los protocolos de cifrado de información, no se pueden utilizar las técnicas clásicas de clasificación. En este Trabajo de Fin de Máster (TFM) se propone utilizar redes neuronales convolucionales para clasificar tráfico, además se propone utilizar FPGAs (*Field Programmable Gate Array, Matriz de Puertas Lógicas Programable en Campo.*) para realizar el proceso de forma eficiente y para no afectar a la calidad percibida por los usuarios. En la **Figura 3.1** se muestra un caso de uso del sistema.

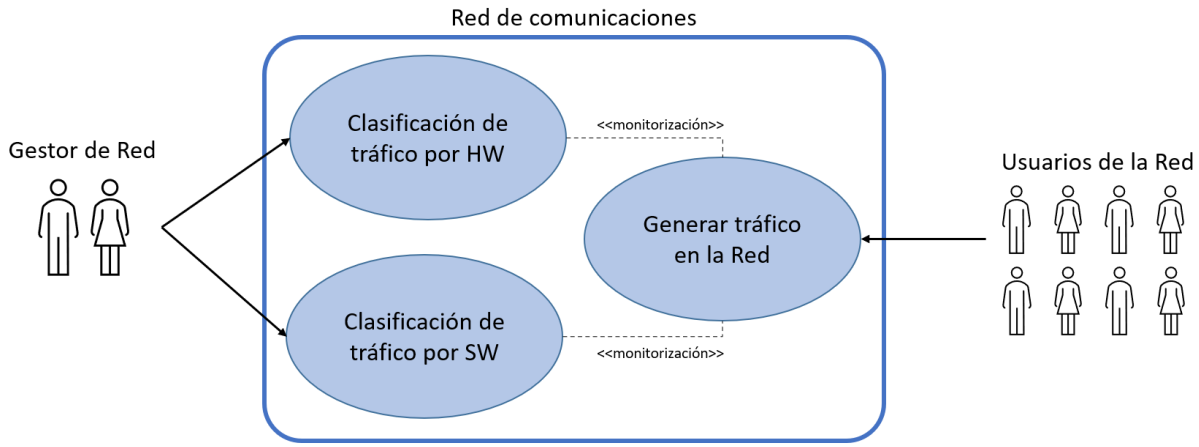

Figura 3.1: Caso de uso del sistema.

3.3. Requisitos funcionales

Para dar por completado el sistema de clasificación de tráfico se proponen una serie de requisitos funcionales, para validar el funcionamiento del sistema y poder determinar que el proyecto ha finalizado adecuadamente.

El principal requisito que debe cumplir, es ser capaz de clasificar paquetes de tráfico de red. Al ser un problema de clasificación n-clases, elegir aleatoriamente las clases nos asegura un acierto total de aproximadamente de 1/n, por lo que el sistema desarrollado deberá mejorar este resultado. Según la literatura actual algunos sistemas alcanzan tasas cercanas al 90 % de acierto en promedio de clases. [13] [45] [19] También se debe asegurar que el proceso de análisis de tráfico y conversión de paquetes a imágenes para usar en la red convolucional es correcto, ya que el trabajo se fundamenta en este proceso.

Otro requisito necesario es que se muestren los valores de los resultados de forma clara y concisa, de esta forma se podrá visualizar el funcionamiento del sistema. Se deberá comprobar el aprendizaje del modelo de red neuronal y comparar los valores de precisión obtenidos entre clases. También se propone que los resultados puedan generar registros o algún tipo de sistema de procesamiento en lote para mejorar la productividad en el desarrollo.

En cuanto al sistema FPGA, deberá ser capaz de procesar las mismas imágenes que el sistema de procesamiento GPU. Con el objetivo de avanzar el conocimiento en este campo, se debe documentar el proceso de instalación y desarrollo en esta arquitectura. Además, se deben diseñar estrategias para la mejora continua del sistema y así poder validar su utilidad en un entorno real.

3.4. Requisitos no funcionales

Una vez se ha diseñado y desarrollado el funcionamiento del sistema, se centrarán los esfuerzos en los requisitos no funcionales. Entre estos, se debe intentar que el sistema alcance tasas de precisión lo más altas posibles.

También, se propone optimizar todo lo posible el código, al estar trabajando con Python en una aplicación de tiempo real, puede resultar crucial para el uso del sistema. Además, se debe documentar el código todo lo posible para ayudar con el posible desarrollo posterior. Otro de los motivos de optimizar el sistema es minimizar la latencia mediante la bajada de los tiempos de procesamiento. De esta forma, se podría utilizar el desarrollo como base para la clasificación de tráfico en redes de alta tasa de tráfico.

También se debe trabajar en la sencillez de uso del sistema, crear un *pipeline* de datos intuitivo y estructurado. Como se van a utilizar distintas arquitecturas de redes neuronales y con distintos valores de entradas. Se propone utilizar una variable que defina los píxeles y la dimensión de las imágenes de entrada a la red neuronal. También se usará otra variable para definir que tipo de red neuronal se va a entrenar, si de tipo CNN o de tipo CNN+LSTM. Con esto, se quiere ayudar en el desarrollo y en las pruebas de testeo para que sean más rápidas de ejecutar y desarrollar. En cuanto a la instalación del sistema en FPGA, se requerirá un sistema sencillo que ayude a la utilización del mismo. Por ese motivo se ha elegido Pynq, ya que permite el uso de una interfaz web mediante *notebook* de Jupyter que resulta fácil de utilizar.

3.5. Conclusión

En este capítulo se ha dado una descripción del problema que se desea solventar. Para ello, se han especificado los requisitos que debe cumplir el clasificador de tráfico para llevar a cabo su función. Estos requisitos se usarán para establecer los objetivos a cumplir en el diseño y desarrollo, y permitirán verificar el cumplimiento de dichos objetivos propuestos para el Trabajo Fin De Máster.

4
Diseño

4.1. Introducción

En el capítulo anterior se ha realizado una revisión del estado del arte sobre la clasificación de tráfico cifrado. En este capítulo, se va a explicar el diseño inicial planteado para el clasificador de tráfico. Primero, se va a explicar el conjunto de datos etiquetado que se ha utilizado, que contiene tráfico de red cifrado de VPN (*Virtual Private Network, Red Privada Virtual*) y otras aplicaciones. Después, se va a explicar, con la ayuda de un diagrama de flujo de trabajo, los procesos que permitirán desarrollar el sistema de clasificación. Por último, se describirá el entorno de desarrollo que se ha utilizado en este proyecto, incluye los sistemas operativos utilizados, los entornos de desarrollo, los lenguajes de programación utilizados en cada tarea y las especificaciones *hardware* de los equipos utilizados.

4.2. Conjunto de datos

El conjunto de datos (o *dataset*) elegidos es el *dataset* de experimentación ISCXVPN2016 [46]. Este *dataset* se compone de 25 GBytes de capturas de tráfico dividido en 152 archivos en formato `.pcap`, estos se agrupan a su vez en 42 categorías de par aplicación-servicio. Por ejemplo, tiene distintas aplicaciones de *video-streaming*, como Youtube o Netflix, pero también tiene capturas de Facebook audio, vídeo y chat. Por último, este *dataset* tiene varias capturas a través de VPN, que aseguran que el tráfico está cifrado. En la **Tabla 4.1** se muestran las distintas aplicaciones que contiene el *dataset* según el tipo de tráfico y su contenido. Es uno de los conjuntos de datos más estudiados en el campo de la clasificación de tráfico cifrado.

En cuanto al filtrado y procesado de los paquetes, muchos estudios omiten por completo el proceso de adaptación del *dataset* al problema, explicar este apartado es crucial

Tabla 4.1: Aplicaciones que contiene el *dataset*. [18]

Traffic	Content
Web Browsing	Firefox and Chrome
Email	SMPTS, POP3S and IMAPS
Chat	ICQ, AIM, Skype, Facebook and Hangouts
Streaming	Vimeo and Youtube
File Transfer	Skype, FTPS and SFTP using Filezilla and an external service
VoIP	Facebook, Skype and Hangouts voice calls (1h duration)
P2P	uTorrent and Transmission (Bittorrent)

para no llevar a cabo errores en la metodología, en este proyecto se comentará en el capítulo 5. Previamente a procesar los paquetes, se deben descartar aquellos que sean de puertos bien conocidos (DNS o DHCP, por poner algún ejemplo), también se deben descartar aquellos que no incluyan información que se deba clasificar, como los paquetes ACK, SYN de TCP que no contengan datos.

La primera parte de desarrollo de este Trabajo Fin de Máster consistirá en procesar este *dataset* para obtener imágenes. Estas imágenes deberán ser representativas de los paquetes de tráfico de cada aplicación y se debe separar los flujos para luego poder entrenar y testear el modelo. Cabe comentar que los datos de *Web Browsing* no se han utilizado, pues no están etiquetados en el *dataset* descargado. Tampoco se ha podido utilizar los archivos P2P (*Peer To Peer*), pues las capturas de tráfico no se podían procesar por uso excesivo de memoria en el ordenador.

4.3. Flujo de trabajo

En la siguiente **Figura 4.1** se muestra el flujo de trabajo realizado para este proyecto. En ella también queda documentado el entorno de desarrollo utilizado en cada etapa, así como las entradas, salidas y formatos de cada proceso.

El proyecto se ha dividido en tres fases, diferenciadas por colores en la figura. La primera fase consiste en el procesamiento de los datos de captura de tráfico para convertirlos en un conjunto de datos en imágenes para utilizar en la red neuronal convolucional. Estos datos se utilizarán para entrenar el modelo en la segunda etapa, también se procesará el modelo para adaptarlo al formato DPU. Por último, se compararán los resultados obtenidos en la DPU con los de la CPU + GPU.

La primera actividad de la primera fase (la azul), es la descarga y estudio del conjunto de datos. Se ha elegido el ISCXVPN2016 [46], se ha comentado en el apartado anterior el motivo. Después se debe filtrar, en este proceso se utilizará tshark, que es una potente herramienta de captura y procesamiento de tráfico. La entrada de este proceso será una captura en formato .pcap o .pcapng y la salida será el mismo archivo en formato .pcap filtrado. Después las capturas de tráfico filtradas se procesarán con un analizador de paquetes en C, que se utilizará para descartar las cabeceras de los paquetes, agrupar los paquetes por flujos y guardarlos como archivo binario en formato `.bin` para cada paquete. El último proceso de la primera fase, será convertir cada archivo binario a una imagen de la dimensión y tamaño que se desee. Se utilizará Python, en un *script* desarrollado

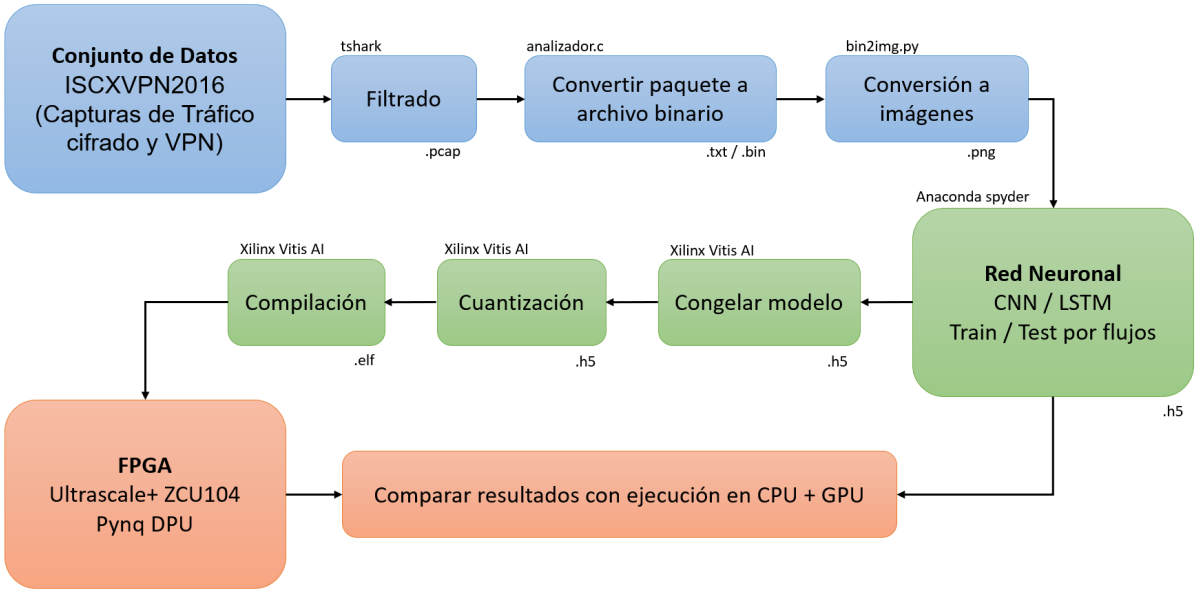

Figura 4.1: Flujo de trabajo.

previamente en [47]. La entrada de este proceso será cada paquete en formato binario `.bin` y la salida será una imagen en formato `.png`.

Para la segunda fase (en verde) consiste en entrenar el modelo de red neuronal y obtener un modelo ejecutable en la DPU. La primera actividad será la más duradera y compleja de este proyecto, consiste en generar un *dataset* de entrenamiento y testeo a partir de las imágenes que obtuvimos de la fase anterior, después estos datos se usarán para entrenar varios modelos de redes neuronales y realizar pruebas de precisión. Se utilizará el entorno Anaconda Spyder con Python y Tensorflow Keras. La entrada será un conjunto de datos en formato `.png` y la salida será un modelo de red neuronal en formato `.h5`. Los siguientes procesos serán: congelar el modelo, cuantizarlo y compilarlo. Los tres se realizan en el entorno Vitis AI de Xilinx, y las salidas son en los dos primeros casos un modelo en formato `.h5` y en el caso de la compilación un archivo binario ejecutable en la DPU con el formato `.elf`. Más adelante en la memoria, en el apartado 5, se comentará que debido a ciertas dificultades, se ha bajado la versión de Vitis AI utilizada y el archivo ejecutable tendrá formato `.xmodel`.

La última fase del trabajo (en naranja) consiste en realizar un código Python en un *Notebook* de Jupyter, para ejecutar el modelo obtenido en el apartado anterior en la DPU. Después se realizarán medidas de rendimiento y se compararán los resultados con la ejecución del mismo modelo en un entorno CPU + GPU.

4.4. Entorno de desarrollo

Como se ha visto en el apartado anterior, este proyecto se compone de varias fases diferenciadas, tanto en el ámbito teórico como técnico. Para el desarrollo de cada una de estas partes se van a utilizar distintas aplicaciones en dos sistemas operativos. Se han elegido según las ventajas que ofrecen de cara al desarrollo y al conocimiento personal que se tiene de estas.

Primero se ha utilizado una máquina virtual con Ubuntu, para desarrollar los *scripts* y el programa en C que procesará los archivos de captura de tráfico. El programa en C utilizará la biblioteca *libpcap* para analizar las cabeceras de los paquetes. Otros usos serán la utilización de *scripts* en *bash*, principalmente se usará para filtrar con Tshark el tráfico de puertos conocidos y procesar las capturas en diferentes carpetas, lo que ayudará a organizar las imágenes en flujos de red. También se proponen algunos test de validación utilizando *scripts* de *bash*. Luego, se usará Python con la librería PIL, que permite generar imágenes PNG para usar como datos en la red neuronal.

Aparte de la máquina virtual en Ubuntu, se utiliza un sistema Windows con tarjeta gráfica para desarrollar el modelo de red neuronal y testeo. Las especificaciones del equipo son las siguientes:

- **CPU**: Procesador Intel i5 6500, 4 núcleos, frecuencia 3.2 - 3.6 GHz, caché inteligente de 6 MB.
- **GPU**: Tarjeta gráfica Nvidia GTX 1060 6 GB de VRAM
- **Memoria RAM**: 16 GB de memoria DDR4 a 2400 Mhz
- **Disco duro**: SSD
- **Sistema Operativo**: Windows 10, 64 bits, Versión 1909
- **Entorno de desarrollo**: Anaconda Spyder,

Este equipo tiene instalada la plataforma de desarrollo Anaconda, que es bastante usada en el campo del *deep learning* y *data science*. En esta plataforma se han instalado un entorno de Python con *TensorFlow 2.3.1 & Keras 2.4.0*. Además, el paquete de *TensorFlow* instalado es el de GPU, que permite utilizar la potencia de paralelización de las tarjetas gráficas para mejorar el rendimiento de las redes neuronales.

Por último, se ha utilizado una máquina virtual Ubuntu que tenía instalado el *docker* de Xilinx Vitis AI 1.3. Esta máquina virtual se ha utilizado también para realizar la conexión con la FPGA. La FPGA utilizada es una Zynq ZCU104 Ultrascale+, a la cual se le ha instalado un *overlay* de Pynq DPU v2.6 con versión de la DPU v3.3. Para acceder a la DPU se ha utilizado *ssh*.

4.5. Conclusión

En este capítulo se han comentado el diseño que se ha realizado para el clasificador de tráfico cifrado propuesto. Lo primero que se ha explicado ha sido el conjunto de datos utilizado, las clases de tráfico que contiene. También se ha explicado el proceso que se debe realizar al conjunto de datos para obtener un *dataset* de imágenes que pueda alimentar el modelo de red neuronal. Esto se ha explicado en el apartado de Flujo de trabajo, en este apartado se explica que entradas y salidas tendrá cada fase del proyecto, así como el formato y el entorno utilizado.

Por último se ha detallado el entorno de desarrollo utilizado para cada fase del proyecto, que incluye una máquina virtual con Linux y Windows. También se incluye el entorno de desarrollo VitisAI para obtener un modelo de red neuronal ejecutable en DPU.

Este diseño se deberá seguir en el apartado de Desarrollo para realizar el sistema de clasificación y así poder obtener resultados que validen su funcionamiento.

5 Desarrollo

5.1. Introducción

En este capítulo se va a explicar las fases llevadas cabo en el desarrollo del clasificador de tráfico. Se usará el análisis de requisitos del capítulo 3 como objetivo a cumplir y se seguirá el diseño planteado en el capítulo 4.

Posteriormente, se explicará cómo se han procesado las capturas de tráfico para poder obtener datos de calidad para entrenar la red neuronal. Se tendrá en cuenta la importancia de simular con estos datos un entorno de red real. Luego, se describirá la arquitectura de la red neuronal y los elementos que la componen, también se explicarán que pruebas se han realizado para obtener los mejores valores de precisión.

Finalmente, se comentará la implementación en *hardware* que se ha realizado con el entorno de desarrollo de VitisAI y las dificultades que han surgido al trabajar con la DPU de la FPGA.

5.2. Filtrado de paquetes y adaptación a imágenes

Al analizar los archivos `.pcap` se puede observar que estas capturas contienen varios protocolos distintos. Por ejemplo: DNS que permite traducir URL (*Uniform Resource Locator*, Localizador de Recursos Uniforme) a direcciones IP (*Internet Protocol*, Protocolo de Internet); ARP (*Address Resolution Protocol, Protocolo de Resolución de Direcciones*) que permite resolver direcciones en la capa de enlace; DHCP que permite asignar direcciones IPs de manera dinámica, o NTP (*Network Time Protocol*, Protocolo de Tiempo en la Red) para sincronizar los relojes de los equipos.

Estos protocolos no envían contenido que interese clasificar para caracterizar el tráfico, por ese motivo debemos descartarlos, pues teóricamente serán protocolos que queden en la red interna de la empresa o el cliente y no deban ser procesados por el ISP. En caso

de que tenga que procesarlo, serán clasificados rápidamente porque son protocolos que utilizan puertos bien conocidos.

Posteriormente, cuando se obtenga un conjunto de datos con los protocolos que se desean clasificar, se deberá procesar las capturas de tráfico para adaptarlas a un formato imagen que pueda usarse en la red neuronal. También se debe guardar la información en carpetas de forma que queden organizados por aplicaciones y flujos de la captura de tráfico.

5.2.1. Filtrado de los archivos de capturas de tráfico

El primer *script* que se ha desarrollado es muy sencillo de entender. Se llama *script_filtrar.sh* y lo que hace es recorrer todos los archivos `.pcap` descargados y los filtra utilizando Tshark. Tshark es la versión en terminal de línea de comandos del popular disector de tráfico Wireshark. Este programa permite procesar los archivos a alta velocidad además de soportar archivos de varios GBytes de tamaño. El filtro aplicado a los archivos es el siguiente:

```
"tcp && tcp.len != 0 || ssl || gquic || data && not dns"
```

Este filtro lo que hace es quedarse únicamente con los paquetes de tipo TCP (*Transmission Control Protocol*, Protocolo de Control de Transmisiones), SSL (*Secure Sockets Layer*, Capa de Puertos Seguros), GQUIC (*Quick UDP Internet Connections*, Conexiones UDP Rápidas en Internet) o UDP, pero no DNS. Los paquetes TCP son aquellos que contienen información de transmisión confiable. Los paquetes de SSL son aquellos que van cifrados para garantizar una comunicación segura. Y los paquetes de GQUIC son una capa de transporte especial que utiliza Google para sus servicios, funciona por encima de UDP. Una de las ventajas de comenzar aplicando este filtro es que permite reducir el tamaño del conjunto de datos de 28 GBytes a algo menos de 9 GBytes. En la siguiente **Figura 5.1** se observa una comparación en Wireshark de una captura tras ser procesada con el *script* de filtrado. Como se observa, se han eliminado los paquetes ACK (*Acknowledgement*, Asentimiento) y los ARP. La reducción de paquetes es bastante grande porque hasta un 25 % del tráfico de red se debe a paquetes ACK vacíos.[48]

Una vez tenemos los paquetes de tráfico de aplicaciones, debemos procesarlos para poder adaptarlos después a una imagen. Este proceso se hará en dos partes, primero se procesará el archivo y se convertirán los paquetes de tráfico a archivos binarios. Después, se tomarán esos archivos binarios y se reorganizarán en píxeles para formar una imagen. Es importante estudiar qué tamaño y dimensión se desea en las imágenes. Para este proyecto se ha decidido usar imágenes 1D de 784 píxeles, 1024 píxeles y 1444 píxeles e imágenes de 2D de 28x28 píxeles, 32x32 píxeles y 38x38 píxeles.

El programa utilizado para convertir los archivos `.pcap` en binarios se ha llamado *analizador.c*, está programado en lenguaje C, que permite procesar rápidamente los archivos `.pcap` gracias a la integración de la biblioteca *libpcap*. El proceso es simple, pero la ejecución tiene algunos detalles relevantes:

Lo primero que se debe hacer es compilar mediante el comando *gcc -Wall -o analizador analizador.c -lpcap*. Hay que asegurarse de que el archivo *analizador.h* está en la misma carpeta. Con esto, tendremos el ejecutable listo. Este ejecutable tiene varios argumentos de entrada importantes: `-f` para seleccionar el archivo (*file*) que se va a analizar y la

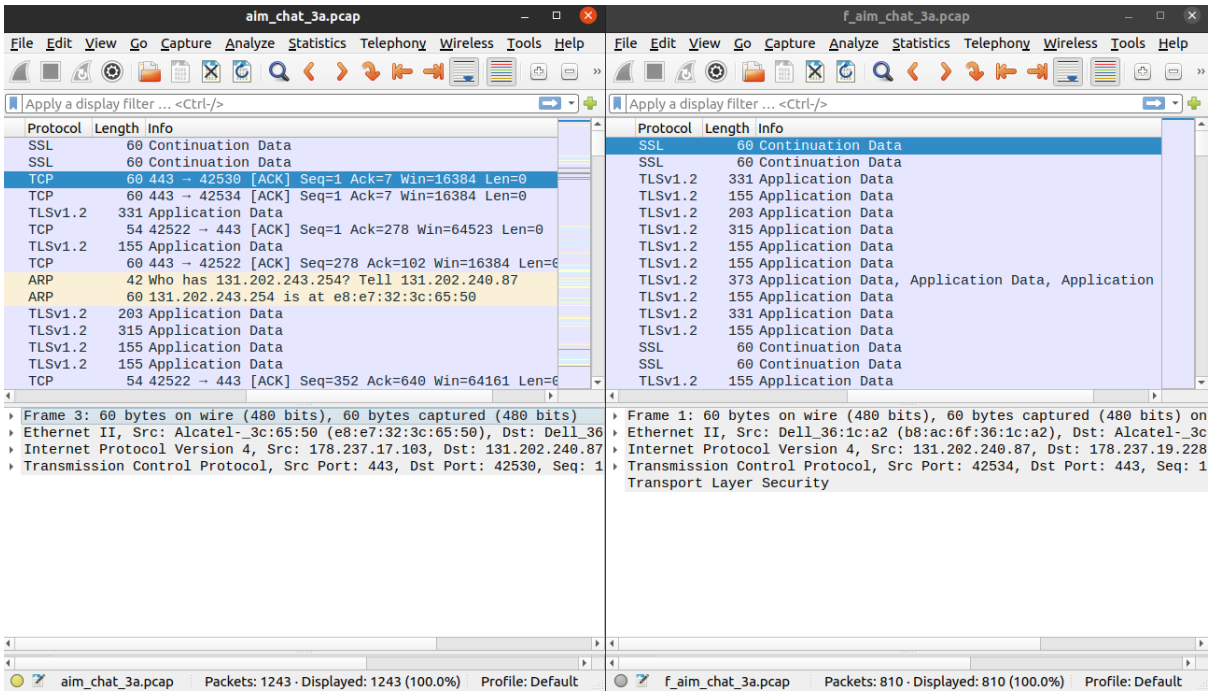

Figura 5.1: Comparación en Wireshark de una captura tras el filtrado.

opción `-e` para añadir un filtro extra al analizador. Por ejemplo, filtrar por una dirección IP concreta o protocolo concreto. Para este trabajo, esta opción no es necesaria. Por último, comentar que tiene una opción *legacy* de cuando se desarrolló un analizador en el Trabajo Fin de Grado [49], (`-i`) que sirve para capturar desde una interfaz de red. No se ha probado para este caso, pero para un desarrollo posterior podría resultar útil.

Para ejecutar el analizador sobre un archivo debemos usar el comando: `./analizador -f fichero.pcap`. Para no tener que poner este comando para los 152 archivos `.pcaps` se ha diseñado un *script* que facilita la tarea, se llama `script_pkt_to_bin.sh`. Lo que hace es mover un archivo `.pcap` a una carpeta con el mismo nombre y ejecutar ahí el analizador, de esta forma se puede tener organizado por carpetas todos archivos de tráfico separados por las distintas capturas.

En cuanto al funcionamiento interno del analizador, este procesa los paquetes de la captura uno a uno, y en orden de llegada de los paquetes al archivo pcap. Estos los analiza utilizando los campos conocidos de Ethernet, IP, UDP y TCP. También se guardan los datos de IP origen y destino, puerto origen y destino y tipo de transporte, con esos cinco datos, también llamados quíntupla, se puede agregar el tráfico por flujos o sesiones. Esto permite ordenar por subcarpetas de flujo los paquetes. Para convertir los paquetes a archivos binarios se utiliza la función *fprintf()* y se recorre *byte* a *byte* los datos de cada paquete y se escriben en un archivo binario (formato `.bin`).

Cabe mencionar que los datos correspondientes a las cabeceras Ethernet, IP y TCP deben de ser descartados por varios motivos: el primer motivo, es que no aportan información de los datos de la aplicación del usuario, por lo tanto, no se estaría haciendo un clasificador de tráfico según las aplicaciones; el segundo motivo, es que pueden generar fallos en el modelo de red neuronal, ya que podría aprender a diferenciar una aplicación por utilizar una dirección IP concreta, un puerto concreto o un valor de ACK o SYN. Esto no es necesario en el modelo de red neuronal, ya que un sistema de puertos conocidos

aprovecharía de forma más eficaz la información de puertos y evitaría el sobre-ajuste. Además, la información de ACK o SYN no tendría por qué ser necesaria para clasificar tráfico.

Por último, comentar que cuando un paquete no llegue al tamaño mínimo se deberá rellenar los datos que falten. Se proponen dos modelos: el primero, sería rellenar con ceros el resto del paquete; el segundo, sería rellenar con valores aleatorios. Para este trabajo se ha utilizado el primer método, ya que permite ver claramente donde termina la información del paquete y esta información podría ser utilizada por la red neuronal. En desarrollos futuros puede ser interesante estudiar el segundo modelo, ya que evita el sobre-ajuste y permite comprobar que la red está aprendiendo correctamente mediante características espaciales. Resulta interesante comprobar como algunos estudios recientes se muestran escépticos con respecto al uso del Aprendizaje Automático en la clasificación de tráfico cifrado. Estos consideran que utilizando las métricas de IP, puertos y tamaño del paquete se pueden obtener resultados similares en precisión sin depender de una arquitectura compleja y lenta como son las redes neuronales. [50]

Una vez se han filtrado los paquetes el conjunto de datos obtenido tendrá las siguientes propiedades: el tamaño total del *dataset* es de 3 Gbytes, está repartido en 149 carpetas con las capturas de tráfico del conjunto de datos original. Dentro de cada una de estas carpetas se encontrarán los flujos de tráfico de red. En total hay 26.805 flujos y no están repartidos equitativamente entre las distintas capturas, algunas tendrán más flujos y otras tendrán menos. Por último, para cada dimensión (1D o 2D) habrá en total 362.768 imágenes entre todos los flujos de todas las capturas, de nuevo, no todos los flujos tendrán el mismo número de imágenes pues están repartidas irregularmente.

5.2.2. Convertir paquetes a formato imagen

Para convertir a imágenes los paquetes en formato binario, se ha utilizado un *script* desarrollado por Ignacio Sotomonte en su Trabajo Fin de Máster [47], su *script* convierte los archivos binarios en imágenes PNG de una, dos y tres dimensiones. En este proyecto las imágenes 3D (con colores) no van a ser utilizadas, pues no aumentan ni la velocidad por paralelización ni obtienen mejores resultados de precisión.

Para ejecutar este código Python de forma eficaz se ha creado un *script* en *bash*, llamado *script_ bin_ to_ img.sh*, que se encarga de recorrer todas las carpetas de capturas, todas las sub carpetas de flujos y ejecutar el comando para convertir los archivos a imágenes. Este proceso creará dos subcarpetas más: 1Dimension y 2Dimension. En estas carpetas se encontrarán las imágenes de los paquetes en la dimensión correspondiente. El nombre de las imágenes en la carpeta de cada flujo son según el orden cronológico de llegada en el archivo de tráfico original, no según el orden del paquete en el flujo. Este detalle es mínimo, pero más adelante se comentará un problema de Python al procesar este tipo de nomenclatura.

5.2.3. Estructura de carpetas

Como se ha comentado, la estructura de carpetas es un directorio principal, con una carpeta para cada archivo de tráfico, dentro de estas varias sub carpetas con los flujos de

tráfico de cada archivo y por último dos carpetas más con los archivos en imágenes en 1D y 2D. En la **Figura 5.2** se puede ver un esquema de la organización de carpetas.

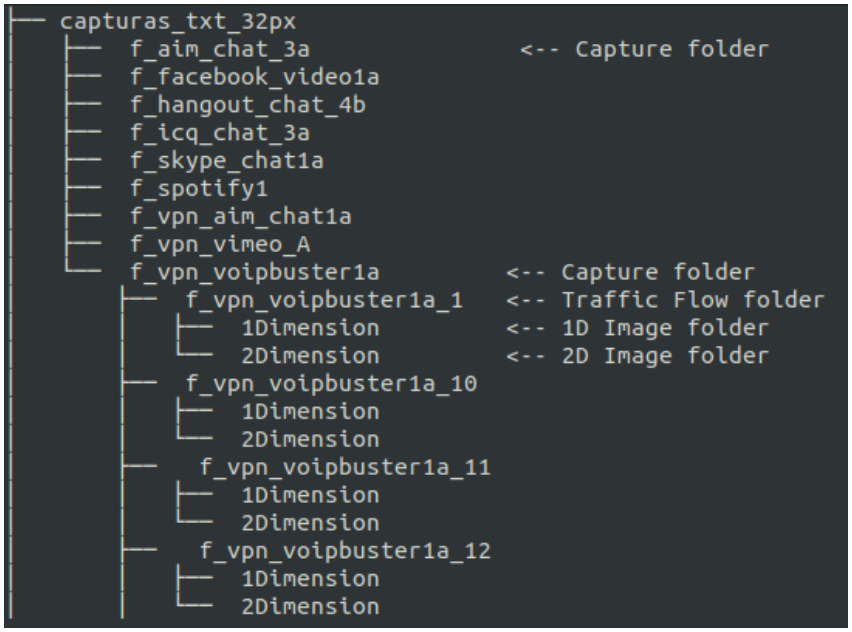

Figura 5.2: Estructura de carpetas según captura de tráfico, flujo y dimensión de la imagen.

Esta estructura se ha elegido así para solucionar dos problemas a la vez: el primero, descarga del programa Python que genera el modelo de red neuronal el trabajo de organizar las imágenes por flujos. Esto permite seleccionar rápidamente qué flujos deben ser de entrenamiento y cuáles de test. Es importante hacer esta diferencia porque como se comentó en el capítulo 2, los artículos leídos no aclaran cómo lo procesan y podría generar un problema de sobre-ajuste, al estar testeando con los paquetes del mismo flujo con el que se entrenaron y que el modelo no funcione en un entorno real. El segundo problema que soluciona esta estructura de carpetas es que permite tener para un flujo las imágenes de una o dos dimensiones en una misma ruta. Esto será útil para hacer pruebas con distintas dimensiones de entrada en la red neuronal. Únicamente se necesitará una variable que actúe como selector de la carpeta de dimensión.

Un problema que genera este sistema de agrupar imágenes por flujos es que en algunos casos, como las aplicaciones de tipo transferencia de archivos, se observarán muy pocos flujos con varios miles de imágenes y luego muchos flujos con muy pocas imágenes, esto se comentará en el apartado de resultados. Una posible solución es limitar cuántos paquetes se utilizan por archivo de tráfico, así se asegura que todos los flujos tienen un número de paquetes balanceado. Sin embargo, se puede estar introduciendo distorsiones en el entrenamiento, ya que no se observará en todos los flujos el cierre de conexión. También es cierto que cada vez más los protocolos optan por cerrar sesiones de forma implícita para liberar memoria rápidamente y evitar enviar más información de la necesaria. Por ejemplo, GQUIC, el nuevo protocolo de transporte de Google, incluye una opción de cierre implícito para evitar enviar paquetes que gestionan el cierre de la comunicación. [51]

Por último, solo queda cargar las imágenes en el directorio de trabajo del entorno de Python. En este caso, se ha elegido un Sistema Operativo Windows con una instancia

de Anaconda Environment en la que se ha instalado Keras 2.2.4 y TensorFlow 2.3.1. Sin embargo, el sistema de procesamiento de archivos de tráfico se había realizado en un sistema operativo Linux (Ubuntu) en un ordenador distinto. Para enviar las imágenes al ordenador de desarrollo del modelo, se ha comprimido la carpeta raíz de imágenes (la que divide por capturas de tráfico) y se ha copiado al ordenador con Windows. Hay que asegurarse de que el formato de compresión sea compatible en Windows y Ubuntu, por ejemplo, ZIP.

5.3. Clasificación de imágenes

Una vez se han procesado los archivos de tráfico, se obtiene un conjunto de datos de imágenes que podemos utilizar para entrenar una red convolucional. Todo el código del sistema de clasificación se realiza en Python. Este lenguaje suele ser preferido a otros con respecto al procesamiento de datos y librerías de *Machine Learning* como Keras y TensorFlow. Además, se tenían conocimientos previos de dicho lenguaje, por ese motivo se ha elegido el mismo.

Lo primero que se ha hecho es instalar *tensoflow-gpu*. De esta forma, se ha configurado el entorno para poder ejecutar la red neuronal utilizando la potencia de la GPU. Se debe comprobar que se ha configurado correctamente mediante el comando `tf.test.gpu_device_name()`. Este debe mostrar *"Default GPU Device: /device:GPU:0"*. En el capítulo de 4 se ha detallado el equipo que se ha utilizado para desarrollar y testear el sistema de clasificación de tráfico.

Otra configuración previa que se debe realizar es el sistema de lectura de las etiquetas de las clases. Dado que esta información no esta adjunta a las imágenes, se ha realizado una lista con el nombre de todas las carpetas generadas a partir de los archivos de tráfico. Al lado de esta lista se ha establecido la etiqueta de clase correspondiente según el nombre del archivo de captura de tráfico. Debido a esto, no se han encontrado referencias a las capturas de buscadores web como Google Chrome o Mozilla Firefox que menciona el conjunto de datos [46]. Otra clase que no se ha podido procesar es la de los archivos P2P, llamados *vpn_ bittorrent* y *Torrent01*. Estos archivos, al igual que alguno de descarga de archivos, resultaban demasiado pesados para procesar con el equipo utilizado. En el caso de los archivos FTP no se ha perdido la clase pues había otros archivos más ligeros para utilizar, pero en el caso de P2P sí se ha perdido la clase.

5.3.1. Separar flujos de entrenamiento y de testeo

Como se ha comentado en el capítulo 2 algunos de los estudios leídos no aclaran cómo se organizan los datos de entrenamiento y test. Para este Trabajo Fin de Máster se ha realizado una separación de los datos de tráfico en flujos. De esta forma, un flujo que se utilice para entrenar no se utilizará en el test, evitando con esto el sobre-ajuste y proponiendo un modelo más acorde con una red real.

Este proceso se adelantó en la parte de generar las carpetas según los archivos de tráfico web. Dentro de cada carpeta están guardadas varias subcarpetas con los flujos asociados a esa captura de tráfico concreta. Para poder dividir los flujos entre entrenamiento y test se proponen tres métodos:

Supongamos que deseamos dividir el conjunto de datos entre 70 % archivos de entrenamiento y 30 % archivos de testeo. En el primer caso, se asigna el primer 70 % de los flujos al entrenamiento, el último 30 % se asigna al conjunto de datos de test. La ventaja de este método es que resulta fácil de implementar y de depurar. La desventaja principal es que los flujos de entrenamiento serán los primeros de la captura y en caso de haber relación entre los flujos se estaría introduciendo cierto sobre-ajuste.

El segundo método propuesto es realizar un sistema de *round-robin* entre los flujos. De esta forma, los primeros 7 flujos serían de entrenamiento, los 3 siguientes de test, de nuevo 7 de entrenamiento etc. Este método corrige el posible sobre-ajuste introducido debido al orden en los flujos.

El último método propuesto consiste en guardar aleatoriamente el 70 % de los flujos en entrenamiento y el 30 % en test. Este método resulta muy difícil de depurar y comprobar errores. A cambio, no habrá problema en el orden de flujos y su relación y sobre-ajuste.

El método elegido para este TFM ha sido el segundo, ya que permite depurar el proceso mejor y teóricamente evita el sobre-ajuste. Sin embargo, las pruebas realizadas no indican que haya diferencia en la precisión según el método utilizado.

Un problema que se deriva del uso de flujos en el entrenamiento, es que no se podrá tener control sobre el número de imágenes y paquetes que usará el conjunto de datos de entrenamiento y test. Esto se debe a que el número de paquetes por flujo no puede ser controlado por el sistema. Esto puede resultar en sobre-ajuste debido a descompensar el número de imágenes por clases. La solución a este problema es poner un límite máximo de paquetes para entrenar y testear igual para todas las clases. Esto implica que algunos flujos no cargarán todas sus imágenes y se quedarán a la mitad.

Durante el desarrollo del sistema de reparto entre los datos de entrenamiento y test, se detectó un posible error que no se había considerado cuando se generarón las imágenes. Estas están guardadas según el orden de captura del tráfico. Por lo tanto, el primer paquete será `1.png`, el segundo `2.png`, el tercero `3.png`, etc. Por lo tanto, si observamos las imágenes dentro de la carpeta de un flujo deberíamos observar unos valores parecidos a: `1.png, 4.png, 6.png, 11.png, 13.png`, etc. El problema detectado es que Python no ordenará las imágenes en orden ascendente, sino que buscará aquellas que empiecen por 1 primero, luego las de 2, etc. Entonces, el flujo anterior la función `os.listdir()` de Python lo ordenará de la siguiente manera: `1.png, 11.png, 13.png, 4.png, 6.png`. Para solventar este error, se ha desarrollado una función llamada `sorted_alphanumeric()` que se asegura de que el orden de las imágenes sea el correcto. [52]

Por último, para prevenir el sobre-ajuste después de cargar las imágenes se deben procesar para ajustar los valores de los píxeles a una normalización de la imagen. En este caso la normalización viene hecha del *script* que convierte los paquetes a imágenes, ya que solo utiliza escala de grises, y los valores quedan contenidos entre 0 y 255.

5.3.2. Estructura de la red convolucional

En este Trabajo Fin de Máster se propone utilizar una arquitectura de red convolucional para clasificar las imágenes creadas a partir de paquetes de tráfico cifrado. Se han generado dos modelos equivalentes para trabajar con imágenes en 1D y 2D.

El objetivo de trabajar con imágenes de distintas dimensiones es poder comparar las ventajas e inconvenientes que presentan. Podemos anticipar que las imágenes bidimensionales serán procesadas más rápido, debido a la paralelización de las GPUs. Sin embargo, perderán algo de precisión debido a que introducen una dimensión que originalmente no existe (un paquete de tráfico se lee linealmente de principio a final).

Antes de detallar el modelo de red neuronal propuesto se deben realizar algunos comentarios sobre las capas utilizadas.

La capa *Dropout* se utiliza para congelar los pesos de neuronas elegidas aleatoriamente durante el entrenamiento. Este proceso dificulta el aprendizaje de la red neuronal, pero a su vez añade complejidad y previene el sobre-ajuste del modelo.

La función de activación *Relu* o rectificador o función rampa, realiza la siguiente función: *f(x) = Max(0,x) para x valor de entrada.* Esto permite a la red neuronal pasar el valor recibido como *input* a la siguiente capa, mientras que si es negativo quedará en 0. Se utiliza para prevenir el problema del desvanecimiento por gradiente en vez de otras funciones tipo *sigmoide* y la tangente hiperbólica. [53]

Cabe decir que, durante la fase de desarrollo, se plantearon distintos valores de filtros, número de capas, valores de *dropout* o distintas funciones de activación. Sin embargo, los valores mostrados coinciden con los utilizados en distintos estudios leídos [19]. También son los valores a los que llegó Ignacio de Diego de Sotomonte [47], el trabajo del cual parte este estudio.

Las capas del primer modelo de red neuronal convolucional propuesto para clasificar imágenes son las siguientes:

1. **InputLayer.** Capa de entrada de las imágenes en 1D, el tamaño en píxeles queda definido por el valor *input_ shape.*
2. **Convolucional 1D.** Capa que realiza la operación de convolución. Tiene espacio dimensional de salida de 32 y cada convolución se realiza con una ventana de 3. La función de activación será la *ReLu function.*
3. **Max Pooling.** Para aumentar la complejidad de las características detectadas por la red neuronal, se debe aumentar la profundidad de las capas convolucionales. Para reducir las dimensiones de la red se utiliza la capa *max pooling* de valor 2.
4. **Convolucional 1D.** Capa que realiza la operación de convolución. Tiene espacio dimensional de salida de 64 y cada convolución se realiza con una ventana de 3.
5. **Max Pooling.** También se utiliza para profundizar en la complejidad del modelo.
6. **Dropout.** Capa *Dropout* para evitar el sobre-ajuste. Valor de *loss rate* de 0.2. Es decir, en cada capa congela el 20 % de las neuronas.
7. **Dense.** Capa densa que interconecta todas las neuronas, tiene un tamaño de salida de 64.
8. **Dropout.** Otra capa *Dropout* para evitar sobre-ajuste. También de valor 0.2.
9. **Dense.** La última capa, y la de salida, es una densa que tiene de tamaño el número de clases. En este caso se deja variable para hacer pruebas con distintos tipos de clasificación de tráfico.

El segundo modelo de red neuronal convolucional propuesto es similar al primer modelo presentado, pero cambian algunos valores para adaptarlo al procesamiento de imágenes bidimensionales. Para adaptar la red a una salida válida se utilizará una capa *Flatten()*. Las capas del modelo son las siguientes:

1. **InputLayer.** Capa de entrada de las imágenes en 2D, el tamaño queda definido por el valor *input_shape* como un *array* de dos valores de píxeles en horizontal y vertical.
2. **Convolucional 2D.** Capa que realiza la operación de convolución. Tiene espacio dimensional de salida de 32x32 y cada convolución se realiza con una ventana de 3x3. La función de activación será la *ReLu function*.
3. **Max Pooling.** Para aumentar la complejidad de las características detectadas por la red neuronal, se debe aumentar la profundidad de las capas convolucionales. Para reducir las dimensiones de la red se utiliza la capa *max pooling* de valor 2x2.
4. **Convolucional 2D.** Esta capa utiliza un valor de *filters* de 64x64.
5. **Max Pooling.** También se utiliza para profundizar en la complejidad del modelo.
6. **Dropout.** Capa *Dropout* para evitar el sobre-ajuste. Valor de *loss rate* de 0.2. Es decir, en cada capa congela el 20 % de las neuronas.
7. **Dense**. Capa densa que interconecta todas las neuronas, tiene un tamaño de salida de 64.
8. **Flatten.** Esta capa transforma los datos de entrada bidimensionales a un *array* unidimensional. Esto es necesario para obtener los valores de etiquetas clasificadas como un *array*.
9. **Dropout.** Otra capa *Dropout* para evitar sobre-ajuste. También de valor 0.2.
10. **Dense**. La última capa, y la de salida, es una densa que tiene de tamaño el número de clases. En este caso se deja variable para hacer pruebas con distintos tipos de clasificación de tráfico.

Este modelo descrito es capaz de clasificar imágenes. Los resultados detallados se verán en el capítulo 6. Pero se puede adelantar que el resultado de precisión general es del 65 % para el conjunto de clases. Queda representado en la **Figura 5.3**

5.3.3. Estructura de la red convolucional y recurrente

Otro modelo propuesto por otros investigadores es el uso de redes neuronales recurrentes [19]. Estos modelos deberían de ser capaces de aprovechar las relaciones temporales de secuencialidad que existen entre los distintos paquetes de un flujo de tráfico.

Previsiblemente, en un flujo de tráfico de red, los paquetes deben ser secuenciales y consecutivos. De forma que, al clasificarlos se deberían poder observar varios paquetes con la misma clase seguidos. Para adaptar a la red neuronal de dicha característica se debe añadir una estructura recurrente.

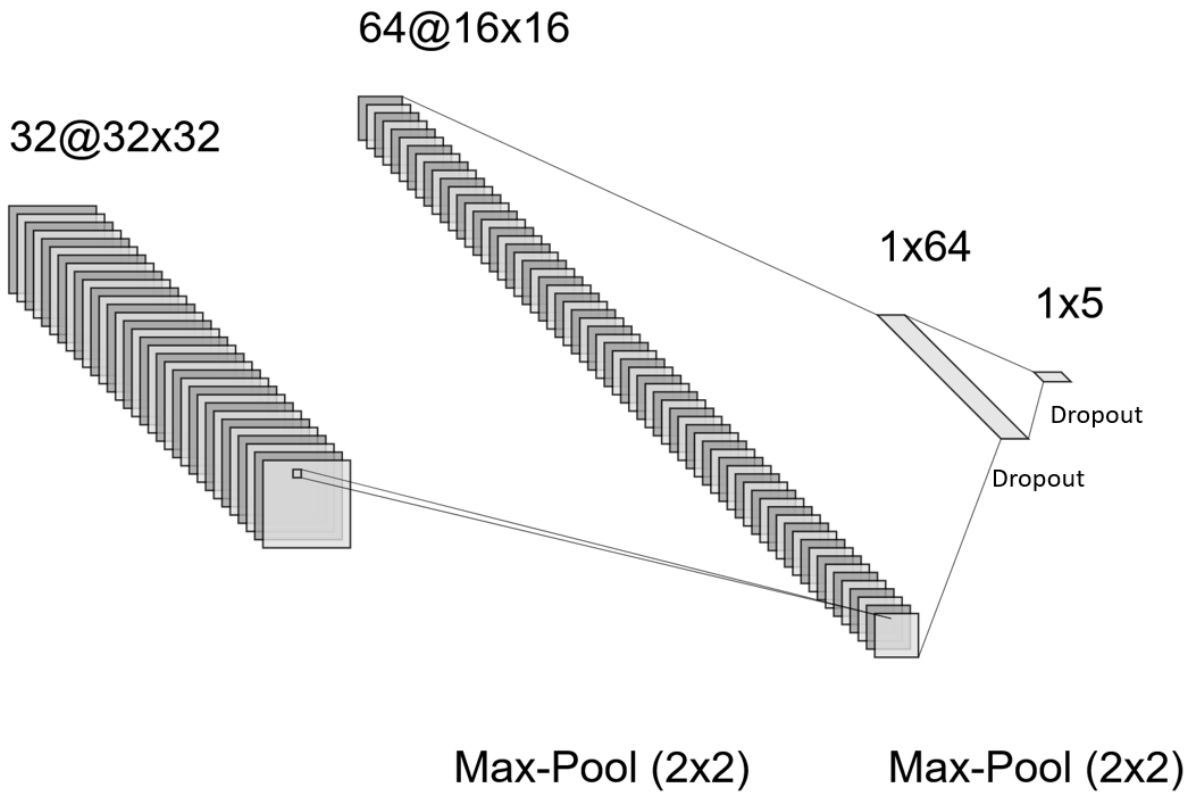

Figura 5.3: Representación del modelo desarrollado con capas convolucionales.

La arquitectura de la red propuesta combina la estructura documentada en el apartado anterior con las capas LSTM que actúan como red recurrente. Las capas del modelo propuesto son las siguientes:

1. **InputLayer.** Capa de entrada de las imágenes en 2D, el tamaño queda definido por el valor *input_shape* como un *array* de dos valores de píxeles en horizontal y vertical.

2. **Convolucional.** Capa que realiza la operación de convolución. Valor de *filters* 32x32.

3. **Max Pooling**. Capa para reducir las dimensiones y aumentar la complejidad del modelo.

4. **Convolucional.** Esta capa convolucional utiliza un valor de *filters* de 64x64.

5. **Max Pooling**. También se utiliza para profundizar en la complejidad del modelo.

6. **TimeDistributedFlatten**. Esta capa es necesaria para adaptar la salida de la capa convolucional a las capas recurrentes.

7. **LSTM**. Valor de *units* 50. Esta capa permite a la red retener en memoria valores de imágenes previas.

8. **Dropout**. Capa *Dropout* para evitar el sobre-ajuste. Valor de *loss rate* de 0.2.

9. **Dense**. Capa densa que interconecta todas las neuronas, tiene un tamaño de salida de 64.
10. **LSTM**. Valor de *units* de 25.
11. **Dropout**. Otra capa *Dropout* para evitar sobre-ajuste. También de valor 0.2.
12. **Dense**. La última capa, y la de salida, es una densa que tiene tamaño número de clases. En este caso se deja variable para hacer pruebas con distintos tipos de clasificación de tráfico.

Con este modelo de red neuronal se espera que el modelo sea capaz de obtener información temporal de los flujos de tráfico, y que el aprendizaje a largo plazo de un grupo de paquetes de red del mismo flujo permita aumentar la precisión del modelo. Queda representado en la **Figura 5.4**

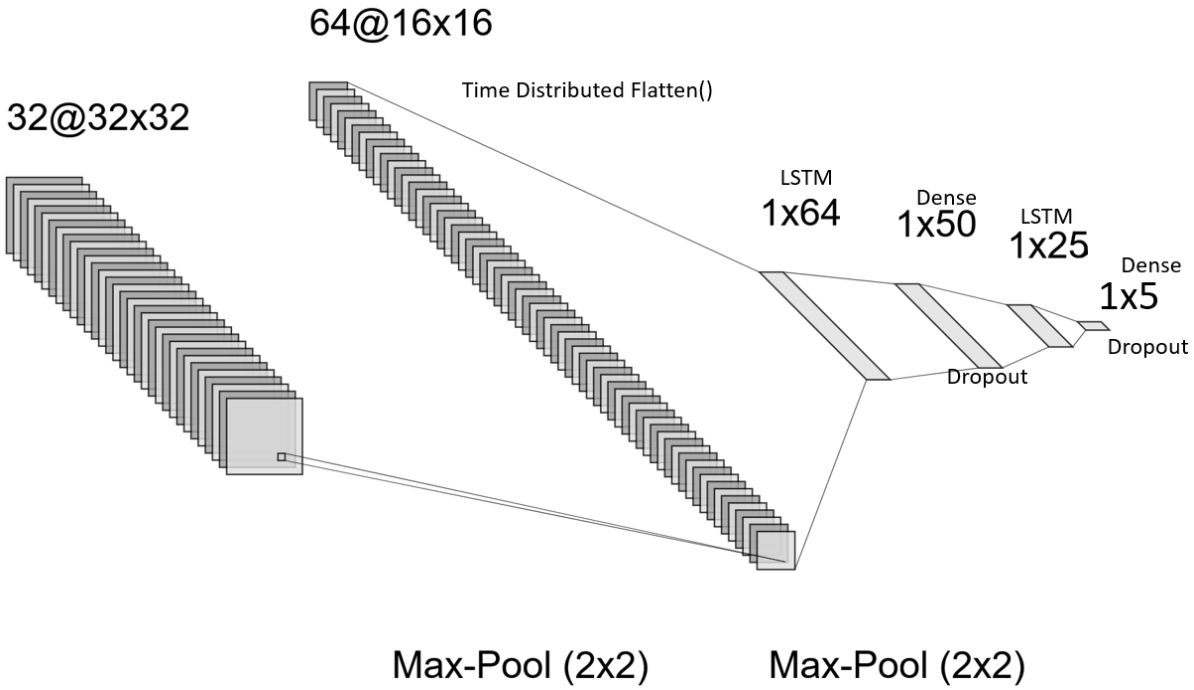

Figura 5.4: Representación del modelo desarrollado con capas convolucionales y LSTM.

5.3.4. Estudio de cómo afecta el tamaño de las imágenes y la dimensión

Otra propuesta de este proyecto es encontrar qué relación existe entre el aprendizaje de la red neuronal y las dimensiones de las imágenes. También se estudia cómo afecta el tamaño de dichas imágenes a la precisión del modelo.

En primer lugar, se estudia cómo afecta la dimensión al rendimiento del clasificador de tráfico. Según los resultados de otros estudios, trabajar con menos dimensiones puede ayudar a obtener valores de precisión mayores. Sin embargo, aumentar las dimensiones permite el uso de arquitecturas paralelas que permiten procesar a tasas mayores, aunque

a cambio se pierde ligeramente precisión en el modelo. En principio, se puede deber a que se crea una dimensión nueva que no existe originalmente en el paquete de tráfico, donde la información es secuencial y no tiene por qué estar relacionada una línea de píxeles con la inmediatamente superior o inferior.

En otros estudios, se han considerado arquitecturas 1D, 2D y 3D. Con imágenes 3D se refiere a con gama cromática RGB, no escala de grises. Los resultados mostrados sugieren que para 2D y 3D los resultados son similares [47]. Por ese motivo, se excluye del estudio las imágenes tridimensionales.

Para el caso del tamaño de las imágenes se plantean 3 modelos, para imágenes de una dimensión 784 píxeles, 1024 píxeles y 1444 píxeles. Como se puede observar, con este número de píxeles se pueden convertir a imágenes bidimensionales cuadradas. Es decir, de 28x28 píxeles, de 32x32 píxeles y de 38x38 píxeles. Los valores de estos píxeles se obtendrían de los primeros bytes del paquete una vez se han descartado las cabeceras (menos la de aplicación) y si el paquete no tiene suficiente tamaño se rellenan los siguientes píxeles en negro. Esto quiere decir, que en ningún caso se reescalan las imágenes para adaptarlas a un tamaño menor o mayor, si se hiciese eso se estaría indirectamente manipulando el contenido del paquete de tráfico.

Los valores escogidos no son aleatorios, provienen de responder al siguiente planteamiento: ¿Cuántos bytes de un paquete se deben procesar para maximizar el acierto del modelo?. Como sabemos, en una red Ethernet el valor MTU (*Maximum Transmission Unit*, Máximas Unidades de Transmisión) es 1500 bytes. Es decir, no debería haber paquetes transmitidos que ocupen más de 1500 bytes. Por lo tanto, si queremos maximizar el valor de bytes en una imagen 1D, y que sea raíz de un valor entero, debemos usar 1444 o 1520 bytes. Pero si usamos 1520 bytes todas las imágenes tendrán 20 bytes sin datos, o sea, los últimos 20 píxeles en negro. Para evitar problemas se utiliza 1444, que en el peor caso se perderán 56 últimos bytes, en un caso típico de IP y TCP se perderían solo 16 bytes. En la **Figura 5.5** se observa el histograma obtenido de todos los paquetes de las capturas, y en la **Figura 5.6** se muestra tras realizar el filtrado.

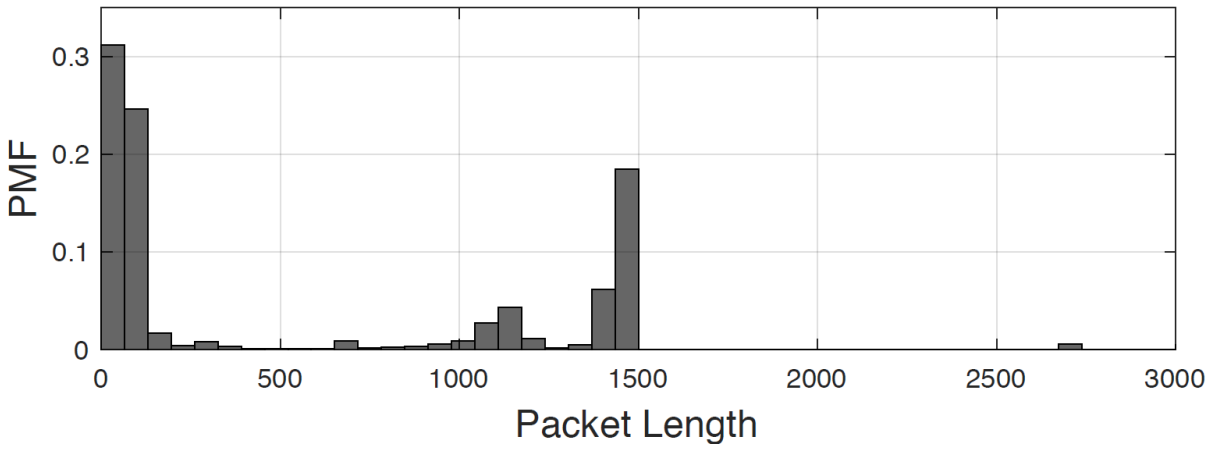

Figura 5.5: Histograma de longitud de paquetes en el conjunto de datos. [13]

En ambos histogramas se observan paquetes cuyo tamaño supera la MTU de una conexión Ethernet. Esto se puede deber a que la captura de tráfico se haya realizado desde los equipos del cliente, al no haber llegado el paquete a la tarjeta de red, puede que

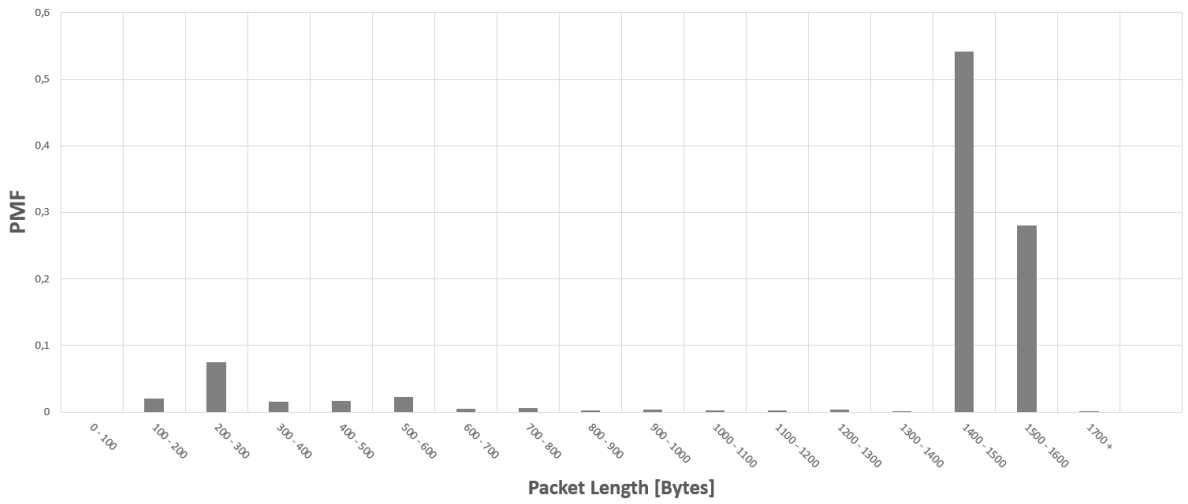

Figura 5.6: Histograma de longitud de paquetes en el conjunto de datos filtrado.

haya algún paquete que no ha sido dividido. Pero es seguro que una vez entra en la red deberá de ser ajustado a la MTU [54].

Con 1444 píxeles podemos formar imágenes de 38x38 píxeles. Ahora bien, si estudiamos la distribución de densidad de todos los paquetes del *dataset* según su tamaño, ver **Figura 5.5**, podemos observar que la media queda cerca de 750 bytes, esto se debe a que hay muchos paquetes que maximizan el valor de MTU, pero también hay muchos paquetes que usan muy pocos bytes, menos de 100. Probablemente se deba a que hay muchos paquetes que solo se utilizan como ACKs o *keep-alives*. Con ese valor de 750, el valor más próximo que sea raíz cuadrada de un número entero es, aproximadamente, 784. Con estos valores tendríamos las imágenes de 28x28 y 38x38 píxeles. Sin embargo, se observa que un gran número de paquetes no maximizan la MTU, pero sí son cercanos a 1000 bytes. Para considerar estos paquetes se ha elegido 1024, o sea 32x32 píxeles. El valor de 32 es un múltiplo de 2 que además coincide con las arquitecturas clásicas de *hardware*. Es previsible que este valor pueda hacer uso de las optimizaciones *hardware* que implementan las CPU, GPU y DPUs.

En el 6 se detallarán los valores obtenidos para los distintos modelos. En cada modelo se estudiarán los distintos valores de entrada combinados con las distintas dimensiones.

5.3.5. Estudio de cómo desordenar los flujos de tráfico

En este trabajo, se ha intentado adaptar el sistema a un entorno realista. Esto quiere decir que las decisiones tienen que ser tomadas pensando en cómo se verían los datos en una red real, como la que pueda verse en alguna empresa, institución o un ISP.

Si observamos una red de tráfico real, los paquetes de los distintos flujos se entremezclan. Es decir, primero veremos unos pocos paquetes de una dirección, luego otros paquetes de otra dirección distinta, luego otra dirección más, tal vez volvamos a ver paquetes de conexiones previas. Esto se debe a que cuando se genera un flujo de datos de larga duración, pensemos en un *streaming* de vídeo, descarga de archivo pesado o llamada VoIP. Los paquetes se acabarán separando y mezclando con los de otras aplicaciones y clientes.

Por ese motivo, en el apartado de generación de los datos de entrenamiento y test, se ha decidido implementar una función opcional, que haga que los paquetes de distintos flujos se entremezclen, pero deben mantener el orden dentro del mismo flujo. Se presupone que el orden o no de los paquetes debe resultar indiferente a la red convolucional. Sin embargo, a la red LSTM, debido a la información temporal que guarda en la memoria, debería traducirse en algún incremento o disminución de la precisión del modelo.

5.4. Aceleración en FPGAs

Uno de los objetivos de este trabajo es desarrollar un sistema capaz de trabajar a alta tasa de procesamiento y baja latencia. De esta forma, un gestor podría implementarlo de cara a realizar una clasificación del tráfico a tiempo real en una red moderna. Los sistemas actuales de CPU + GPU pueden alcanzar altas tasas de procesamiento en paralelo. Por lo tanto, ese objetivo quedaría cumplido con el diseño actual. Sin embargo, hay un detalle importante que se debe considerar.

Para poder alcanzar las tasas óptimas del sistema CPU + GPU, se deben procesar los paquetes en *batch*, es decir, en lotes. Esto en sí no es ningún problema, pero implica, que en un sistema real se debería esperar al último paquete del lote para poder clasificar. Esto quiere decir que en el peor caso, el primer paquete del lote tendrá una latencia adicional igual a la suma de tiempos de llegadas de todos los paquetes anteriores.

En las pruebas realizadas, este valor se puede ir hasta los 130 ms en total. Si imaginamos un paquete de una aplicación de VoIP, de los 150 ms totales que se consideraría una latencia asumible en VoIP, solo tendríamos 20 ms disponibles para procesar en los enrutadores, codificación y decodificación y transmisión de la información físicamente. Por supuesto, estos valores no pueden ser controlados por el gestor de la red. Por lo tanto, no es asumible un sistema con tanto retardo de procesamiento.

Para solventar este problema, se propone el uso de arquitecturas basadas en FPGAs. En estas, se puede realizar un procesamiento casi como en *bare-metal* de forma independiente por imagen. De esta forma, se generará un pipeline, en el cual la latencia máxima de procesamiento de la red neuronal será independiente para cada imagen. Y no se necesitará de esperar a la llegada del resto de paquetes del lote.

La FPGA que se ha utilizado en este proyecto es una Zynq Ultrascale+ ZCU104, facilitada por el laboratorio HPCN-UAM. Este equipo está compuesto por un MPSoC (*Multi-processor System-On-Chip, Sistema en Chip*). En la **Tabla 5.1** se incluye las características del dispositivo:

Para el desarrollo se ha realizado conectando la ZCU104 al *router* de la red local y se ha realizado el desarrollo del sistema en remoto desde ssh y conectando mediante el navegador a la dirección `http://pynq:9090/tree/pynq-dpu`. Para realizar la conexión se debe utilizar la siguiente guía [15]. Los componentes de la FPGA se pueden observar en la **Figura 5.7** y son los siguientes [10]:

- **Configuration**:

  USB-JTAG FT4232H, Dual Quad-SPI flash memory, MicroSD Card

- **Memory**:

Tabla 5.1: Tabla de características de Zynq UltraScale+ MPSoC. [10]

System Logic Cells (K)	504
Memory	38Mb
DSP Slices	1,728
Video Codec Unit	1
Maximum I/O Pins	464

PS DDR4 64-bit Component, Quad-SPI flash, Micro SD card slot

- **Control & I/O**:

  4x directional pushbuttons, DIP switches, PMBUS, clocks, and I2C bus switching, USB2/3

- **Expansion Connectors**:

  FMC LPC (1x GTH), 3 PMOD connectors, PL DDR4 SODIMM Connector – 64 bit

- **Communication & Networking**:

  USB-UARTs with FT4232H JTAG/3xUART Bridge, RJ-45 Ethernet connector, SATA (M.2) for SSD access

- **Display**:

  HDMI 2.0 video input and output (3x GTH), DisplayPort (2x GTR)

- **Clocking**:

  Programmable clocks, System clock, user clock: Jitter attenuator

- **Power**:

  12V wall adaptor or ATX

Por último, en la **Figura A.1** del Apéndice se muestra la visión general de la arquitectura de la Ultrascale+ ZCU104. Como se comentó en el capítulo 2, para poder ejecutar un modelo de red neuronal en una FPGA, se deben seguir unos procesos que permitan reducir el tamaño de esta y convertirla a instrucciones ejecutables por la FPGA.

5.4.1. Cuantización del modelo

Cuando se genera un modelo de una red neuronal, hay muchos datos que se deben guardar para poder ser ejecutada. Por ejemplo, para cada neurona hay que guardar los

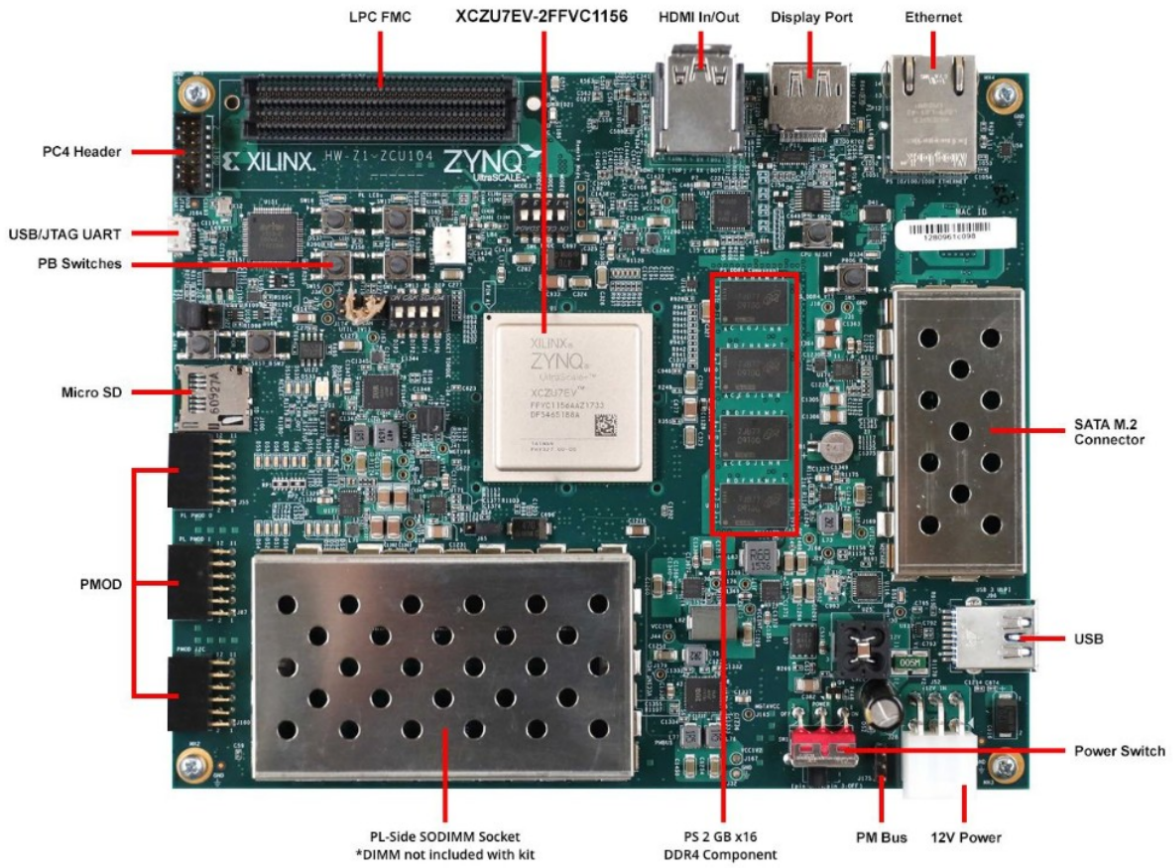

Figura 5.7: Componentes IO de la FPGA ZCU104. [10]

pesos de cada una de los miles de conexiones que tiene. Además, hay que guardar la topología de la red neuronal, los optimizadores que se han utilizado y el estado de los mismos. Por lo tanto, un modelo completo entrenado tendrá un peso considerable.

Para poder optimizar el tamaño de estos modelos y que puedan caber en una memoria de FPGA, además de que sean más rápidos de ejecutar, se debe realizar un proceso de cuantización. La cuantización es el proceso mediante el cual se reducen los bits usados en guardar los pesos de los tensores. Por lo general, un modelo de red neuronal en Keras/TensorFlow guardará los valores en coma flotante. Estos permiten guardar los pesos en un amplio rango de valores y debería dar lugar a modelos más precisos. Sin embargo, la aritmética de coma flotante es mucho más sofisticada que la de coma fija, lo que hace que resulten más complejas y lentas las operaciones. Además, reducir los valores de bits permitirá utilizar la FPGA para un mayor número de operaciones simultáneas [55]. En la **Figura 5.8** se muestra un esquema del funcionamiento del cuantizador.

Cabe preguntarse qué ocurre con la degradación de la precisión en el modelo, pues, como es lógico, perder precisión numérica en los pesos de la red neuronal provoca una disminución de la precisión general del modelo al clasificar. Sin embargo, para valores de 8 bits por peso, hay estudios que demuestran que la pérdida es insignificante. Solamente por debajo de 6 bits, la degradación de la precisión puede comprometer los resultados [55]. Además, hay documentados algunos casos que han demostrado que utilizando un menor número de bits se pueden prevenir efectos de sobre-ajuste, dando lugar a un modelo mejor adaptado a una aplicación real [56]. Por último, es importante considerar que este proceso

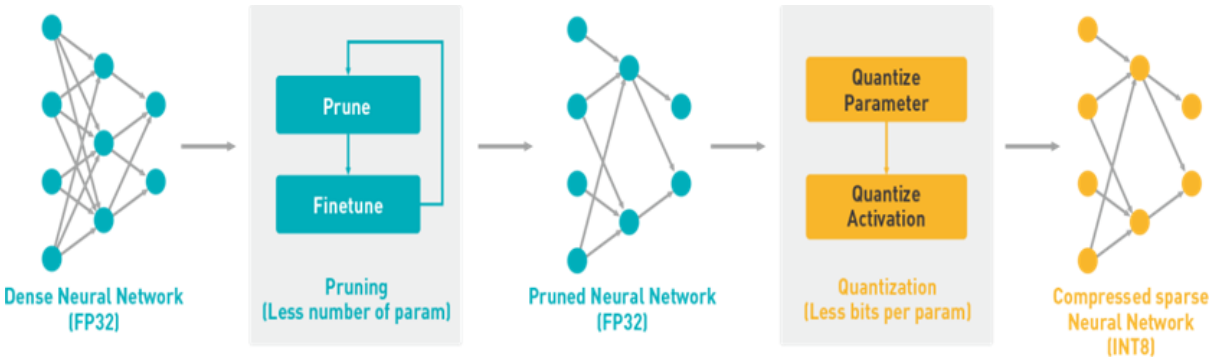

Figura 5.8: Proceso de cuantización de una Red Neuronal. [11]

dará un modelo adaptado a la ejecución en sistemas empotrados, por lo que merece la pena esta cuantización [55].

Para realizar la cuantización es necesario tener un modelo con los valores de las neuronas congelados. A este proceso se le llama *freezing*. Por suerte, en las últimas versiones de TensorFlow (a partir de las 2.4) este proceso resulta automático mediante el uso de la función `model.save()` [57]. Este comando generar un archivo `.m5` que tiene guardada toda la información necesaria para reentrenar o ejecutar el modelo. Este sistema resulta mucho más conveniente que en versiones anteriores de Keras/TensorFlow, donde se generaban tres ficheros separados [55] (más adelante se comentarán cuáles).

Una vez tenemos el modelo en un archivo podemos utilizar la función `vai_q_tensorflow quantize` para ejecutar el comando que cuantizará el modelo. Los argumentos de esta función son los siguientes:

- **input_frozen_graph**. Nombre del modelo que vamos a cuantizar.
- **output_dir**. Directorio donde se guardará el modelo.
- **input_nodes**. Capa que se utilizará como entrada de los datos al modelo.
- **output_nodes**. Capa que se utilizará como salida de los datos al modelo.
- **input_shapes**. Tamaño de los datos de entrada. En este caso, serán imágenes de 32x32 píxeles. Por lo tanto, se pondrá (?,32,32,1).
- **calib_iter**. Cuantas veces se va a iterar el modelo en el proceso de calibración.

Para el proceso de calibración se debe aportar una pequeña muestra de los datos de test. En este proceso no hacen falta las etiquetas, ya que no se va a reentrenar el modelo. Simplemente se comprobarán que las neuronas activadas y desactivadas se corresponden a las de las imágenes de calibración. Para este trabajo se aportaban 100 imágenes aleatorias de los datos de test.

Una vez ha terminado el proceso de cuantización se puede reentrenar la red neuronal para mejorar ligeramente la precisión del modelo. Para ello, haría falta añadir un nuevo conjunto de datos de entrenamiento etiquetado. Este se podría generar a partir de un subconjunto del conjunto de datos de entrenamiento original.

Por último, el modelo puede ser testeado mediante la función *accuracy.eval()*. Esto nos permitirá cuantificar cuanto ha sido la pérdida real de precisión en nuestro modelo, debido al proceso de cuantización. En el capítulo 6 se comenta que la pérdida es aproximadamente 2 %, cayendo del 65 % al 63 % de precisión general.

5.4.2. Compilación del modelo

El siguiente paso necesario para adaptar la red neuronal al entorno FPGA sería realizar la compilación del modelo. Este proceso consiste en optimizar y convertir el modelo en un formato ejecutable adaptado al entorno DPU. Para ello, haremos uso de la función `vai_c_tensorflow()`. En la **Figura 5.9** se muestra un esquema del funcionamiento del compilador.

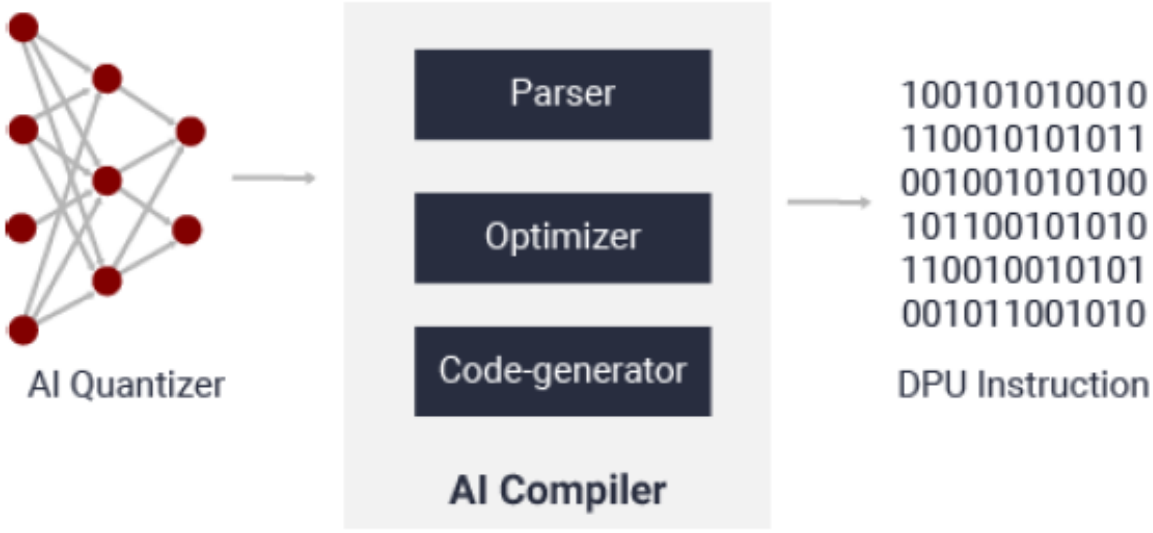

Figura 5.9: Proceso de compilación de una Red Neuronal. [11]

Este proceso consta de varios argumentos:

- **frozen_pb.** Nombre del modelo que vamos a compilar.
- **arch.** Directorio donde se guarda el archivo JSON que guarda la configuración del sistema que va a ejecutar los datos.
- **output_dir.** Directorio donde se va a guardar el archivo de salida.
- **net_name.** Nombre de salida que tendrá el modelo.

En las versiones actuales de VitisAI (2.4) se genera un archivo con formato `.elf`. Este archivo puede ser ejecutado en las nuevas versiones de FPGAs tipo Zynq Ultrascale+ y Alveo. Las nuevas versiones tienen capacidad para ejecutar modelos convolucionales y recurrentes, ya que incluyen la compilación de ambos tipos de capas. Sin embargo, al desarrollar el sistema completo, se ha encontrado que actualmente la capa `TimeFlatten()` no tiene implementación en la versión disponible (**Figura A.3**). En la **Figura A.2** se puede ver qué capas están recogidas en la guía de usuario. Esto quiere decir que, si bien

se pueden ejecutar modelos convolucionales o modelos recurrentes, no se pueden ejecutar modelos complejos que incluyan ambas partes. Como este es el caso de este proyecto, se decidió continuar el trabajo usando únicamente la versión de red convolucional, sin capas LSTM.

Como se muestra en la **Figura A.4** otro problema que se encontró es que los archivos `.elf` generados no los leía correctamente la DPU y daba errores. Por estos motivos, se decidió utilizar una versión anterior de TensorFlow (2.3), Keras (2.4.4), Vitis-AI (v1.2) y DPU (v3.3). Estas versiones anteriores no tenían implementadas las capas recurrentes, pero, como en las actuales tampoco podemos utilizarlas, se decidió probar estas.

En las versiones anteriores de VitisAI los archivos ejecutables que se generaban se guardaban con el formato `.xmodel`. Además, la versión anterior de TensorFlow no guarda los modelos como formato `.h5`, sino que utiliza tres formatos de archivos distintos para guardar la información de todo el modelo. Estos archivos generados tienen formato `.pb`, `.chkp` y `.index`. Estos vienen a partir de las funciones `tf_session.graph.as_graph_def()` y `saver.save()`, que se utilizan para entrenar el modelo con *checkpoints*. Esta herramienta es muy interesante para guardar los parámetros de un entrenamiento y luego recuperarlos para reentrenar redes neuronales en otros modelos o plataformas, sin embargo no profundizaremos en ella (**Figura A.6**).

Una vez se han obtenido los 3 archivos mencionados, se debe congelar el modelo mediante la función `freeze_graph()` que tiene los siguientes argumentos:

- **input_graph** model_classifier.pb Nombre del modelo a congelar.
- **input_checkpoint** `checkpoint.ckpt` Archivo con los valores de pesos del último *checkpoint*.
- **input_binary** *True* Indica que el formato es `.pb`, si fuese `.pbtxt` debería ser *False*.
- **output_graph** `frozen.pb` Nombre de salida del modelo congelado.
- **output_node_names** Nombre de las capas de salida.
- **output_logits_tfm/Softmax** Nombre de las carpetas donde se guardan los archivos.

Para realizar todo el código de ejecución de *tensorflow freeze*, cuantización y compilación, así como las pruebas de precisión, se ha utilizado un *Notebook Python* que se ejecuta en la carpeta `workspace/VitisAI/` dentro del *docker* de VitisAI. Parte del código se ha basado en el ejemplo que propone Xilinx para entrenar un modelo de clasificación de imágenes con el conjunto de datos mnist [58].

Pero volviendo al problema original, con las versiones anteriores de Tensorflow y Vitis AI, ahora se puede compilar el modelo en un formato `.xmodel`, que además acepta el sistema DPU. En el siguiente apartado se comentan los detalles del funcionamiento de la DPU.

5.4.3. Ejecución en DPU

Para conectarse a la DPU se utiliza la dirección local mediante un navegador web, como Google Chrome o Mozilla Firefox desde una máquina virtual. Otra opción es usar una conexión *ssh* desde una máquina virtual con Ubuntu. También hay que subir las imágenes que se utilizarán como conjunto de datos de test. Para esto se ha utilizado una conexión *scp*, es importante obtener los permisos del sistema previamente mediante el comando *chown*.

El código de ejecución de la DPU está basado en uno propuesto por Xilinx para ejecutar una red *Resnet* demo. Lo primero que se ejecuta es la carga de la biblioteca DPU mediante los comandos:

- `from pynq_dpu import DpuOverlay`
- `overlay = DpuOverlay("dpu.bit")`
- `overlay.load_model("modelo.xmodel")`

De esta forma, el modelo de nuestra red neuronal quedará cargado en la DPU y se podrá ejecutar. El siguiente paso es cargar las imágenes de test, para ello se utiliza de nuevo el comando *scp*. Luego se define el directorio de lectura mediante `os.listdir(image_folder)`, de la misma manera que cuando se entrenó el modelo, Python no ordena correctamente las imágenes. Se debe definir la función `sorted_alphanumeric()` para corregirlo y evitar cargar flujos desordenados que impidan realizar medidas de precisión o depurar el sistema [52].

Después, se deben definir las funciones de precarga de las imágenes. En el caso de este proyecto, este proceso se realizó anteriormente, pero si en caso de desarrollar el sistema completo se debería realizar. También se ha corregido la función de cálculo de etiqueta predicha que hace Xilinx, ya que siempre predice la clase inmediatamente posterior. Se puede solucionar cambiando el retorno de la función (**Figura A.5**).

Posteriormente, hay que definir las funciones que determinan el tamaño de entrada de las imágenes a la red neuronal, así como la dimensión de salida de las clases. Se puede realizar mediante las funciones:

- `dpu = overlay.runner`
- `inputTensors = dpu.get_input_tensors()`
- `outputTensors = dpu.get_output_tensors()`

También hay que definir una función para ejecutar secuencialmente las imágenes. Esta función realizará la carga de la imagen desde la memoria SD, la preprocesará y luego la ejecutará en la DPU. Para ello, utiliza la función:

- `job_id = dpu.execute_async(input_data, output_data)`
- `dpu.wait(job_id)`

Con los valores de `output_data`, se puede calcular la clase predicha por el modelo y comprobar la precisión del sistema, comparándolo con las etiquetas originales. Por último, se proponen hacer medidas de tiempo de ejecución mediante las funciones `time.time()`. De esta forma, se separa el procesamiento de la DPU en 4 partes; carga de la imagen, preprocesamiento de la misma, ejecución de la red neuronal y cálculo de la etiqueta.

5.4.4. Límites de la aceleración debido a la carga de imágenes desde SD

En el capítulo 6 se comentarán los detalles de los resultados obtenidos. Pero se adelanta que los resultados de latencia son mejores que un sistema clásico, que era lo esperado. Sin embargo, en *throughput* la velocidad de la FPGA queda un poco por detrás lo esperado. Al revisar los valores se ha encontrado que el tiempo de procesamiento se debe principalmente a la carga de imágenes en la DPU. Para realizar pruebas más justas, se propone hacer un disco virtual en RAM.

Este sistema evita que se carguen las imágenes desde la memoria SD, que ya de por sí es lenta de acceder, pues no está pensada para guardar datos en un sistema de procesamiento en tiempo real. Tampoco es eficiente el bus de datos que conecta la DPU con el SD, en un sistema real se haría un puente entre el procesador y la interfaz de red por lo que no haría falta guardar datos en la RAM. La mejor solución que se ha encontrado es crear un disco virtual que una vez cargadas las imágenes sirva para direccionar las peticiones de carga de imágenes. De esta forma, se minimiza el tiempo de carga de la imagen en la DPU.

Lo primero que se debe hacer es dar acceso de edición al usuario Xilinx a la carpeta donde se va a montar el nuevo sistema de archivos. Para ello, podemos utilizar el comando `sudo chown -R username:group directory`. Después ejecutaremos el comando que crea el sistema de archivos en RAM: `mount -t tmpfs -o size=200m tmpfs /opt/data`, hay otra opción que es usar el `ramfs`. La diferencia principal es que el sistema `tmpfs` usará la memoria RAM hasta que se agote y luego pasará a un sistema de `swap`. El sistema de `ramfs` al agotar la memoria RAM dará error. Para nuestra aplicación solo será necesario unos 95 MB del conjunto de datos de test, el sistema DPU tiene hasta 1 GB de memoria RAM libre. Aunque el sistema `ramfs` debería cubrir con las necesidades del proyecto, se utilizó `tmpfs` que resulta una solución más actual. Además, permite ver qué porcentaje del disco virtual se ha llenado mediante el comando `df -h`. [59]

Para ejecutar el proceso de clasificación sobre el sistema de archivos, se deben mover las imágenes a la nueva ruta creada y acceder a los datos ahí desde el *notebook* Python.

5.5. Conclusión

En este capítulo se ha detallado el proceso que se ha llevado a cabo para la elaboración del clasificador de tráfico, teniendo en cuenta la teoría aprendida en el estudio de los sistemas de clasificación mediante redes neuronales y teniendo como objetivos los requisitos propuestos. Se ha explicado el funcionamiento de cada uno de los procesos realizados en el proyecto, que incluyen; el filtrado del tráfico, convertir los paquetes a imágenes y

entrenar un modelo de red neuronal recurrente. Se ha realizado distintos modelos para adaptar el sistema a imágenes de distintas dimensiones y tamaños.

Este proceso se ha repetido para un entorno FPGA, que requiere de los pasos extra de congelar el modelo, cuantización y compilación. Se han tenido problemas para compilar los modelos recurrentes en las versiones actuales del sistema VitisAI. Por lo tanto, se ha utilizado una versión anterior más estable. Por último se ha utilizado el archivo ejecutable obtenido para ejecutar en un sistema DPU de Pynq. Debido al límite que supone la carga de imágenes desde una memoria SD, se ha realizado un sistema de `tmpfs`, que permite cargar las imágenes desde un disco virtual en RAM mucho más rápido que la memoria SD. En el siguiente capítulo se mostrarán los resultados obtenidos.

6 Pruebas y resultados

6.1. Introducción

En el capítulo anterior se documentó el desarrollo realizado para este proyecto a partir del diseño realizado y con los requisitos necesarios para cumplir con los objetivos propuestos.

En este capítulo se expondrán las pruebas realizadas en este TFM y se mostrarán los resultados que se han obtenido. Quedan separados los apartados de pruebas en las redes neuronales del apartado de pruebas en FPGA para organizar mejor estos datos. Con estas pruebas queda comprobado el funcionamiento del sistema de clasificación y también se valorará si este modelo es capaz de resolver el problema planteado y llegar a los objetivos propuestos.

6.2. Pruebas realizadas al preprocesado del conjunto de datos

Primero se ha evaluado el funcionamiento de los *scripts* de procesamiento de capturas `.pcap`. Después se han realizado pruebas para obtener resultados de la clasificación en redes neuronales. También, se ha hecho un estudio de cómo afecta el número de clases, las dimensiones de las imágenes de entrada y el tamaño de las mismas. Por último, se ha comparado los resultados del sistema DPU con el de CPU+GPU.

El primer *script* que se ha desarrollado es el filtro de protocolos. Para comprobar los resultados de este *script* se han visualizado los paquetes en Wireshark. Después de ejecutar el *script* para varias capturas se ha comprobado que los protocolos disponibles son los establecidos. Gracias a esta prueba se encontraron dos errores en el diseño original. El primero, era que hay paquetes TCP que no incluyen información, por ejemplo, ACKs o SYNs, estos paquetes deben ser descartados mediante el uso de `tcp.len != 0`. Es decir,

si el tamaño de los datos de TCP es 0, se filtran. El segundo error, aunque se filtraba para mantener los paquetes UDP con datos los protocolos DNS que van sobre UDP sí aparecían, por ese motivo se filtraron también explícitamente los paquetes DNS.

El segundo *script* desarrollado es el analizador de tráfico, que convierte de paquetes de captura pcap a archivos binarios. Para probar este *script*, se deben comprobar varias cosas: la primera, que el archivo de salida coincide con los bytes del paquete original. Para comprobar esto, se puede utilizar de nuevo Wireshark, que permite visualizar el contenido en hexadecimal de un paquete individual. En la **Figura 6.1** se compara con el archivo en formato `.bin` que se puede abrir con algún editor de texto o un simple comando `hexdump -C`. La segunda comprobación a realizar es que se están generando correctamente los flujos. Para ello, se utiliza la enumeración de los paquetes según el orden de captura. Se puede comprobar, de nuevo, con Wireshark y filtrando la dirección IP y puerto con el de algún flujo. Desde ahí se puede ver si coinciden los paquetes recibidos con el nombre de los archivos binarios. Para comprobar que todos los flujos tienen un número de paquetes correctos, se puede utilizar la utilidad `flow-statistics` que incorpora la herramienta. Para este apartado, es importante descartar los datos de Ethernet, IP y TCP al contar bytes del paquete. Estas cabeceras se descartaban, por lo tanto, no deben aparecer como en la imagen en formato binario.

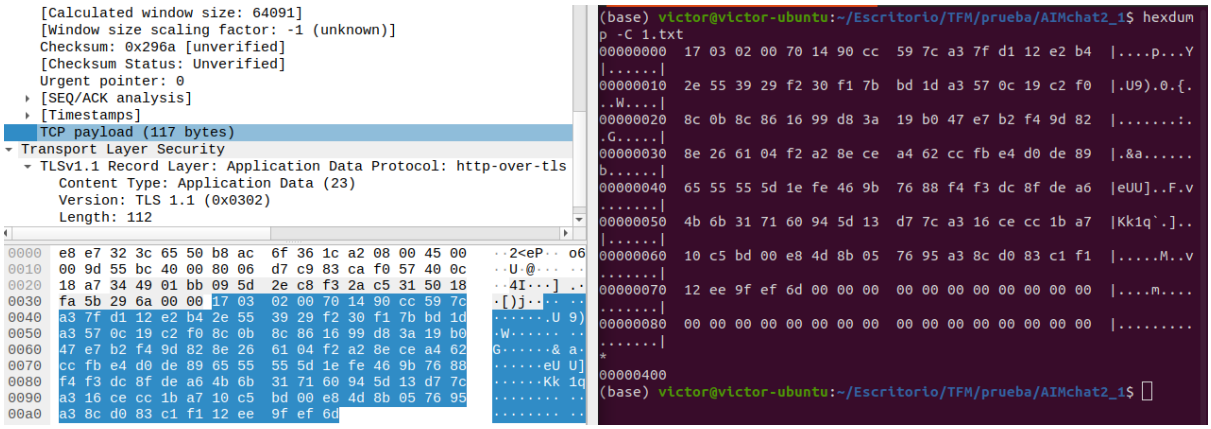

Figura 6.1: Validación del procesamiento del paquete.

El último *script* a comprobar, antes de poner en marcha la red neuronal, es el productor de imágenes PNG a partir de los archivos binarios. Lo primero que se comprueba es que se estén generando imágenes del tamaño correcto. Para ello, se puede utilizar las opciones de detalles que incorpora Ubuntu. No es trivial comprobar que los píxeles coinciden con el valor esperado, pero se puede intentar visualizar en Python la imagen y comprobar que hay píxeles de distinta intensidad hasta cuando terminan los bytes de datos del paquete, el resto deberían rellenarse con píxeles negros. Para ello, se utiliza la función `plt.imshow()` de Python. En la **Figura 6.2** se muestra la imagen estudiada en el apartado anterior. Se puede observar que coincide el número de píxeles con valores. Otra comprobación a realizar es ver si existe la estructura entre paquetes del mismo flujo o clase, ya que esto es en teoría, la motivación de realizar una clasificación por imágenes. Se ha observado que entre ciertos flujos si existe una relación entre los paquetes en cuanto a estructuras y formas, pero esto no es algo general. Tampoco se puede decir, que haya una estructura común a los paquetes de la misma clase. Esto probablemente dificulte el aprendizaje de la red neuronal. Que no exista una estructura espacial que identifique las

clases o que las características temporales de los flujos no sean frecuentes dificultará el aprendizaje del modelo con LSTM. Si bien es cierto, en algunos casos se han encontrado similitudes. Por ejemplo, como se muestra en la **Figura 6.3**, el flujo 11 y 16 de Aimchat tienen paquetes parecidos y en el mismo orden.

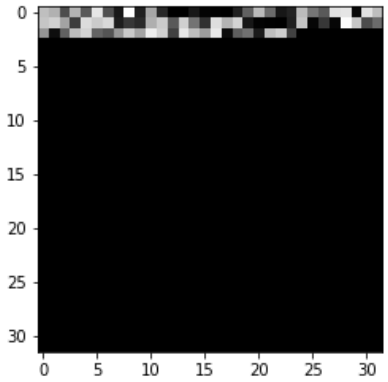

Figura 6.2: Representación de un paquete como imagen 2D.

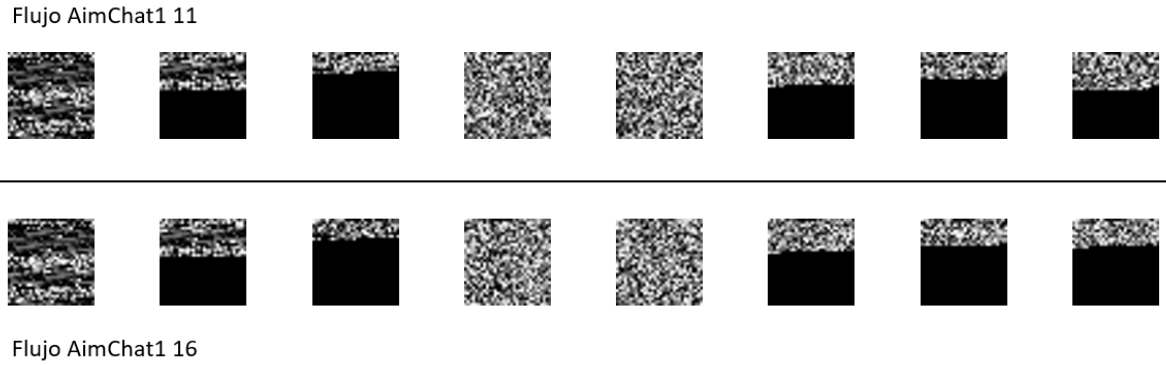

Figura 6.3: Comparación de imágenes de distintos flujos de la misma captura.

6.3. Resultados obtenidos en las redes neuronales

Una vez comprobados los *scripts* de preprocesamiento de las imágenes, se realiza el código de la red neuronal. El modelo creado es difícil de examinar, pues uno de los problemas de las redes neuronales es su falta de transparencia. Lo que se propone es realizar distintas pruebas y comprobar los resultados variando parámetros como número de dimensiones, tamaño de imágenes y número de clases. En el capítulo 5 se ha comentado la motivación detrás de variar las dimensiones y el tamaño de los paquetes. El motivo de variar el número de clases es para comprobar sí resulta más efectivo clasificar en clases generales, sin tener en cuenta la aplicación específica en uso, o si por el contrario, resulta más efectivo clasificar teniendo en cuenta la aplicación concreta detrás del tráfico.

Antes de detallar los resultados obtenidos se describen brevemente las métricas más utilizadas en el campo del Aprendizaje Automático: exactitud (o *accuracy*), precisión (o *precision*), exhaustividad (o *recall*), valor F (o *f1-score*) y soporte (support). Otros términos importantes son los siguientes:

- **Verdadero Positivo:** o TP (por sus siglas en inglés, *True Positive*), son los casos en los que la clase predicha y la real coinciden en SÍ.
- **Verdadero Negativo:** o TN (*True Negative*), son los casos en los que la clase predicha y la real coinciden en NO.
- **Falso Positivo:** o FP (*False Positive*), la clase real es NO y la clase predicha es SÍ.
- **Falso Negativo:** o FN (*False Negative*), la clase real es SÍ y la clase predicha es NO.

Con estos valores podemos definir mejor las métricas: [60]

- **Exactitud:** Es la relación entre las observaciones acertadas y el total de observaciones.
- **Precisión:** Es la relación entre las observaciones acertadas positivas y el total de las observaciones etiquetadas como positivas.
- **Exhaustividad:** Es la relación entre las observaciones acertadas y el total de la clase actual.
- **Valor F:** Es el peso de la media armónica entre la precisión y la exhaustividad.
- **Soporte:** Número de datos de la clase dada.

En la **Figura 6.4** se detalla un resumen de las fórmulas:

Exactitud	(TP+TN)/(TP+FP+FN+TN)
Precisión	TP/(TP+FP)
Exhaustividad	TP/(TP+FN)
Valor F	2*(Precisión * Exhaustividad) / (Precisión + Exhaustividad)

Figura 6.4: Ecuaciones de las métricas utilizadas.

Con estas métricas se evaluó para la ejecución del modelo de CNN (sin LSTM) con las imágenes de 32x32 píxeles, es decir, de dos dimensiones. Lo primero que se comprobó fue cuantas épocas hay que entrenar el modelo para encontrar la convergencia de la precisión. Primero se probaron con 25 épocas, que resulto insuficiente porque aún parecía aprender, se decidió usar 50 para ver si había un límite superior. Al representar las gráficas (**Figura 6.5**) se observa que el valor de exactitud se mantiene en 60-65 % a partir de 15 épocas. Sin embargo, los resultados de test no llegan a converger del todo y los resultados con los datos de entrenamiento parecen seguir aprendiendo.

Para esta ejecución, los resultados son cercanos al 65 %, este valor es menor al 90 % esperado de los artículos leídos. Sin embargo, no se debe a un error en la generación del conjunto de datos, en el modelo utilizado o los parámetros utilizados para entrenar.

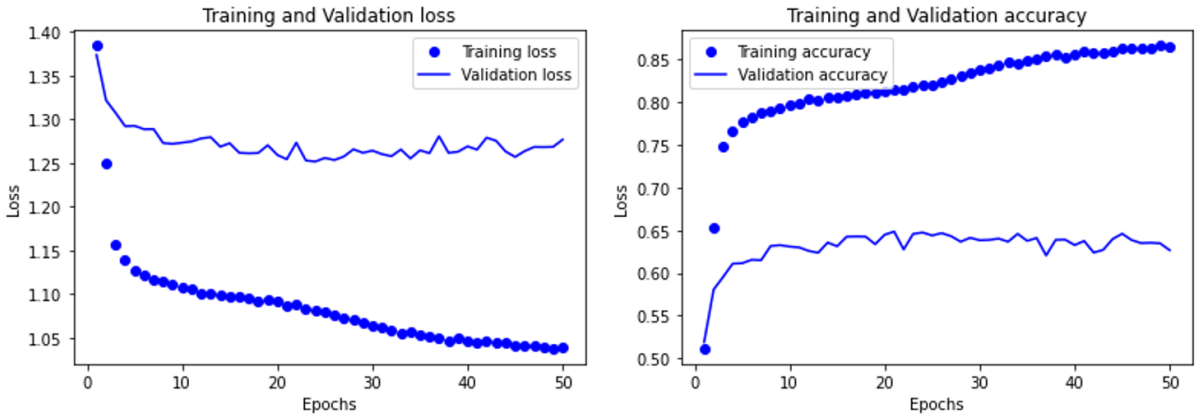

Figura 6.5: Resultados de precisión y pérdida por épocas.

Se debe a que en este proyecto se ha elegido separar en flujos distintos los datos de entrenamiento y de test. De esta forma se evita el sobreajuste y se genera un modelo más cercano a lo que se puede observar en una red real. Esto provoca que las posibles características que existan entre imágenes del mismo flujo (y por lo tanto misma clase) sean más difíciles de encontrar en otros flujos, aunque pertenezcan a la misma clase. Por ese motivo baja tanto el valor de precisión. Además, si se observa el valor de *Training accuracy* en la **Figura 6.5**, se observa como sigue aprendiendo hasta superar la precisión de 90 %.

Esto quiere decir que, con los datos de entrenamiento, al testear sobre ese mismo conjunto, los aciertos son muchos más que en el conjunto de datos de test (que pertenecen a flujos distintos). En este caso sí se obtienen los valores alcanzados por otros autores. Para comprobar que estos valores son correctos, se ha decidido entrenar un modelo con datos que no distinguen entre flujos, es decir, podría haber datos en entrenamiento y en test que pertenezcan al mismo flujo. Como se observa en la **Figura 6.6** se muestra que los resultados son mejores que en el caso en el que se separan los datos por flujos, alcanzando el 90 % de precisión y no hay diferencia en los resultados entre los datos de entrenamiento con los datos de testeo. Esto demuestra la teoría de que para obtener un modelo de red neuronal para clasificar tráfico en una red de tráfico real, se debe entrenar teniendo en cuenta los flujos, de otra forma se estará obteniendo sobreajuste.

Algunos autores, en vez de utilizar solo 5 clases para caracterización de tráfico, utilizan hasta 41 clases distintas para identificación de aplicaciones [47], perteneciendo estas a las distintas aplicaciones y tipo de tráfico. Por ejemplo: Facebook chat, Facebook video, Facebook audio, etc. Básicamente cada par aplicación-servicio se considerará una clase. Se propone estudiar los resultados del modelo con estas clases para comprobar si buscando clases más específicas se pueden obtener mejore resultados.

Al realizar una prueba con este número de clases se observa en la **Figura 6.9** que la precisión del modelo cae sustancialmente al 54 % para imágenes 32x32 píxeles. Si bien es cierto que el objetivo de este trabajo no sería clasificar con mucho detalle los servicios o aplicaciones, sino clases generales que caractericen el tráfico. Se podría plantear comprobar la precisión sobre una clase general de tráfico como puede ser Audio o Vídeo, independientemente de la aplicación que lo genere.

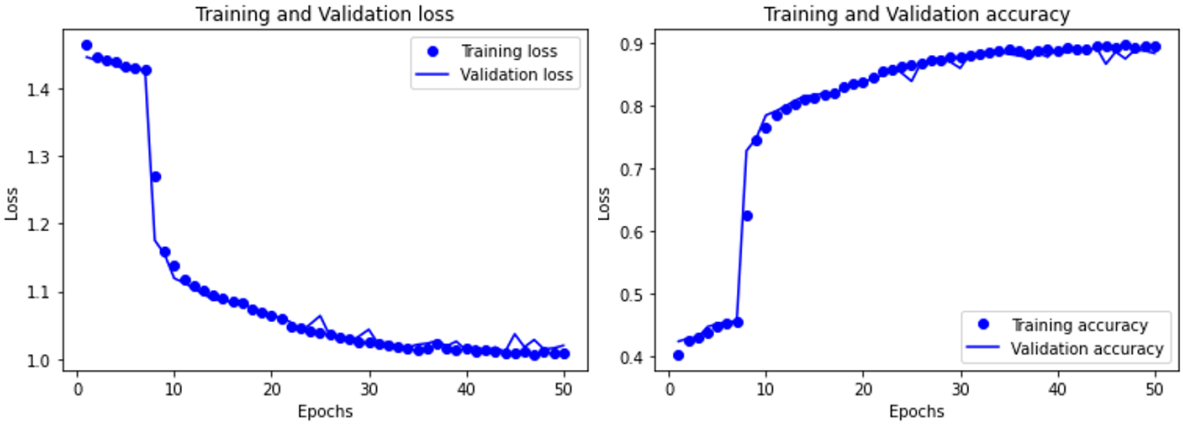

Figura 6.6: Resultados de precisión y pérdida por épocas para datos que no distinguen entre flujos.

Para obtener más información sobre los resultados de la red neuronal, se decide mostrar los datos como una matriz de confusión. La matriz de confusión es una herramienta muy utilizada en el campo de la inteligencia artificial, en especial aplicado al problema de la clasificación. Permite visualizar las predicciones acertadas y falladas por cada clase y de esta forma comprobar qué clases está confundiendo. En la **Tabla 6.1** se muestra el resultado de este modelo. Se observa que la precisión general es cercana al 63 %, pero algunas clases como Filetransfer o Email obtienen un Valor F del 93 % o 72 % respectivamente. Sin embargo, otras clases como Vídeo, Chat o Audio quedan por detrás, obteniendo solamente un 43 %, 53 % y 52 %.

Tabla 6.1: Resultados completos del modelo CNN+LSTM para imágenes 2D y 32px.

	precision	recall	f1-score	support
Audio	0.46	0.63	0.53	4800
Chat	0.58	0.46	0.52	4800
Email	0.62	0.87	0.72	3194
FileTransfer	0.95	0.91	0.93	4800
Video	0.57	0.34	0.43	4800
		accuracy	0.63	22394
macro avg	0.64	0.64	0.62	22394
weighted avg	0.64	0.63	0.62	22394

Observando la **Figura 6.7** vemos la matriz de confusión y se puede observar que la clase Vídeo se confunde bastante con Audio y Chat, la clase de Chat también se confunde bastante con Vídeo. Por el contrario, las clases de Filetransfer o Email apenas confunde paquetes con otras clases. Para encontrar el motivo de estas confusiones se han llevado a cabo tres comprobaciones.

La primera comprobación consiste en revisar los flujos de imágenes de Chat, Audio y Vídeo, para buscar alguna pista de que puede estar causando la confusión. No se encontró ningún motivo por el que las imágenes de Vídeo, Chat o Audio quedasen confundidas. El segundo caso tiene que ver con el balanceo de clases, si bien se ha desarrollado el

	Audio	Chat	Email	FileTransfer	Video
Audio	3014	772	173	79	762
Chat	571	2228	1508	112	381
Email	27	376	2780	1	10
FileTransfer	39	264	43	4385	69
Video	2941	200	7	27	1625

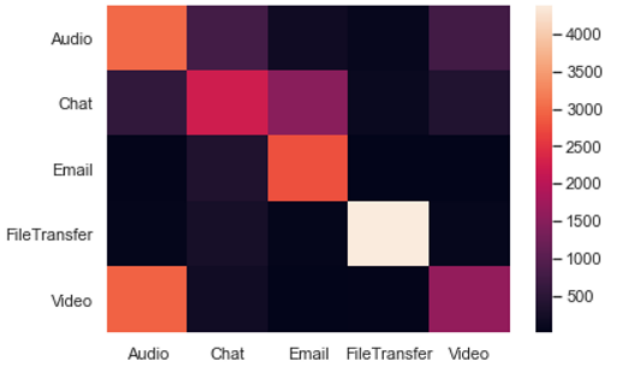

Figura 6.7: Matriz de Confusión de los resultados para el modelo CNN + LSTM para imágenes 2D y 32px.

conjunto de datos de entrenamiento y test con cuidado para evitar el desbalance entre clases, se puede observar que la clase de Email tiene significativamente menos datos. Por ello, se ha decidido reducir el número de paquetes de las otras clases para igualarlo al de la clase Chat. De esta forma, se espera reducir el sobreajuste. Este proceso no es exacto, al usar los paquetes de entrenamiento y test según los flujos el reparto de paquetes no será perfecto y siempre habrá ligeras variaciones.

El último escenario que se plantea para corregir el error es aumentar las clases de vídeo a dos distintas, vídeo en directo y vídeo en descarga (MPEG-DASH). Ya que la primera clase, estará asociada a videollamadas o vídeo-streamings, mientras que la segunda a plataformas de multimedia como Netflix, Vimeo o Youtube. Un problema que se ha encontrado en este conjunto de datos es que no detalla qué tipo de contenido contiene la captura de tráfico y además este no se puede extraer de las capturas. Por ejemplo, Youtube tiene 6 capturas distintas y no es posible encontrar si pertenecen al mismo tipo de vídeo o cambian de formatos entre directos o no. Los resultados obtenidos son similares a los de 5 clases pero algo menores por lo general. Estos resultados se pueden observar en la **Figura 6.9**. Una vez se ha visto que la red neuronal convolucional no mejora los valores de precisión, se plantea el uso de redes neuronales recurrentes y comprobar si mejora la precisión general.

En cuanto a los resultados de la red neuronal con capas LSTM se obtiene también 65 % de precisión, no se observa una clara mejora en los resultados. Es posible que entrenar usando flujos de distintas clases a la vez no sea información suficiente para que la red neuronal recurrente aprenda a clasificar mejor. Al estar los paquetes de los distintos flujos entrelazados, impidiendo que la red identifique la posible secuencialidad temporal de los paquetes.

Para estudiar como afecta desordenar los paquetes entre flujos se proponen dos escenarios: entrenar con paquetes ordenados y entrenar con paquetes desordenados, luego testear ambos métodos con paquetes ordenados y desordenados. Si bien el caso de testeo con paquetes ordenados es irreal pues en una red nunca se debería de dar este caso, resulta interesante como ejercicio teórico y de comparación.

Los resultados obtenidos son los siguientes: para las redes convolucionales hay una disminución del 1 o 2 % de precisión, esto no tiene una explicación directa, pues las redes convolucionales no deberían retener la información sobre el orden en el que están

entrenados o testeados las imágenes. Sin embargo, en la red LSTM, que se debería observar algún cambio, los resultados difieren menos de un 1 %, lo que sugiere que están dentro del margen de error y no hay ningún cambio. Tras los resultados vistos en este proyecto se considera que habría que ampliar la memoria de la red LSTM. En cualquier caso, el juntar varios flujos de tráfico puede generar ruido en la información de la red neuronal, lo cual no ayuda a aprender a clasificar.

	Test	CNN	LSTM
train orden	orden	0.6749	0.6597
	desorden	0.6668	0.6672
train desorden	orden	0.6702	0.6521
	desorden	0.6545	0.6509

Figura 6.8: Comparación entre ejecuciones CNN y CNN + LSTM para tráfico ordenado y desordenado.

Por último, se ha ejecutado el modelo LSTM con los distintos tamaños de imágenes propuestas y distintos números de clases. La **Tabla 6.2** detalla los resultados obtenidos en exactitud general para el modelo CNN+LSTM. Para comprar mejor los resultados se muestra la **Figura 6.9** con los datos de la tabla en forma de gráficas. Por lo general, trabajar con 5 clases da los mejores resultados. Si consideramos las diferencias entre unidimensionales y bidimensionales, las bidimensionales obtienen mejores resultados. Esto resulta contraintuitivo, pues deberían de ser las unidimensionales las que mejor mantienen las estructura del paquete de tráfico. La explicación que se da a este comportamiento es que el modelo de red para imágenes bidimensionales es más complejo, y por lo tanto, ayuda a aprender mejor a clasificar. Por último, las imágenes de más píxeles tienden a tener mejores resultados que las de menos píxeles. Pero, en el caso de las imágenes bidimensionales hay un empate entre 32 y 38 píxeles.

Tabla 6.2: Detalle de resultados de precisión por clases y dimensiones.

	Num clases					
Píxeles	**41**	**5**	**6**	**41**	**5**	**6**
1444	0,61	0,69	0,58	0,63	0,57	0,55
1024	0,44	0,57	0,52	0,53	0,62	0,61
784	0,39	0,55	0,47	0,42	0,55	0,47
38x38	0,49	0,67	0,64	0,51	0,66	0,62
32x32	0,54	0,67	0,59	0,50	0,66	0,61
28x28	0,34	0,55	0,48	0,35	0,56	0,49
	25 epochs 4k pkts por clase			**125 epochs y clases compensadas**		
	70 train 30 test					

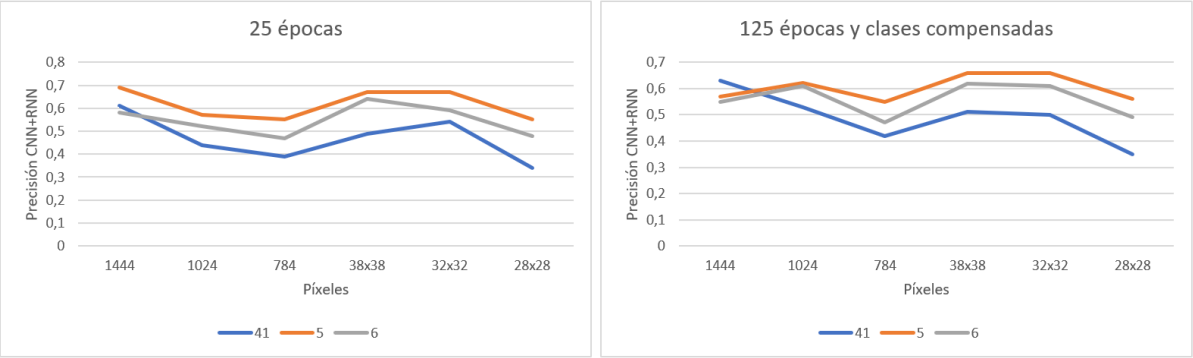

Figura 6.9: Comparación entre ejecuciones CNN + LSTM.

Con estos resultados se propone realizar las pruebas en FPGA utilizando el modelo de red convolucional bidimensional de 32 píxeles. Este modelo es el que mejor resultado ha obtenido junto con el de 38 píxeles. Además, presenta la ventaja de ser potencia de 2, por lo que las arquitecturas hardware puedan estar optimizadas para ello. Y como se comentó en el capítulo 5, la red LSTM no puede ser compilada debido a que falta la capa `TimeFlatten`.

6.4. Resultados de la aceleración en FPGA

El primer test que se ha realizado es la precisión del modelo convolucional compilado, como se explicó en el capítulo 5, es esperable una pérdida ligera de la precisión, en este caso se cumple y el modelo obtiene un 63 % de precisión. Es decir, cae un 2 % frente al modelo sin compilar, que obtenía resultados del 65 % de precisión general.

En la DPU se van a comprobar los valores de tasa de procesamiento y latencia. La tasa de procesamiento es cuántas imágenes es capaz de procesar por unidad de tiempo. Al estar utilizando imágenes, podemos usar la métrica *Frames Per Second*. La latencia será el valor de tiempo que tarda el sistema en obtener una clase para una imagen dada.

Los valores de procesamiento obtenidos para el sistema se muestran en la **Tabla 6.3** comparados con los resultados del mismo proceso ejecutado en el sistema CPU+GPU, en todos casos se han procesado 33.626 imágenes. Para asegurar que los valores son consistentes entre ejecuciones y no se muestra un caso extremadamente favorable o desfavorable se ejecutan tres veces cada prueba y se muestran los resultados de media y desviación típica entre ejecuciones, se muestran los resultados en la **Tabla 6.4**. Además se ha tenido en cuenta la memoria caché para evitar ejecuciones más rápidas debido a este proceso. Sin embargo, más adelante se volverá a comentar.

Para el sistema CPU+GPU se han realizado dos pruebas. La primera consiste en ejecutar una sola imagen cada vez, esto imitaría al sistema de procesamiento de la DPU, que procesa secuencialmente en vez de con lotes. Sin embargo, este sistema en GPU es muy ineficiente y obtiene solamente 26 FPS, pues las GPUs trabajan mejor en paralelo, por ese motivo, la segunda prueba consiste en utilizar un *batch size* de 64 imágenes, para que se procesen en paralelo y se observe el rendimiento en un caso favorable.

Si comparamos los resultados de tasa en el sistema CPU+GPU sin lote, se obtiene una tasa muy baja, de solo 26 imágenes por segundo, la latencia total es de 3,516 s, de los

Tabla 6.3: Resultados de CPU+GPU vs. DPU.

							Por cada img/batch		*En GPU = predict*	
pixeles img	**sistema**	**batch**	**accuracy [%]**	**Performance [FPS]**	**# Img Test**	**Tiempo Total [s]**	**Latencia Total [ms]**	**Latencia Carga imagen [ms]**	**Latencia CNN [ms]**	**Latencia label [ms]**
32x32 px	GPU-CPU	1	65	26,5	64	2,59	3.516,34	0,49		3.515,84
32x32 px	GPU-CPU	64	65	181	33626	186,00	3.609,52	8,49		3.601,02
32x32 px	DPU	1	63	885	33626	38,00	22,88	21,33	0,87	0,68
									Suma:	1,55

							Por cada img/batch		*En GPU = predict*	
pixeles img	**sistema**	**batch**	**accuracy [%]**	**Performance [FPS]**	**# Img Test**	**Tiempo Total [s]**	**Latencia Total [ms]**	**Latencia Carga imagen [ms]**	**Latencia CNN [ms]**	**Latencia label [ms]**
32x32 px	GPU-CPU (caché)	64	65	2457	33626	13,68	3.609,52	8,49		3.601,02
32x32 px	DPU (caché)	1	63	1506	33626	22,33	3,91	2,33	0,90	0,68
32x32 px	DPU (tmpfs)	1	63	1515	33626	22,20	4,01	2,34	0,93	0,74
									Suma:	1,67

							Por cada img/batch		*En GPU = predict*	
pixeles img	**sistema**	**batch**	**accuracy [%]**	**Performance [FPS]**	**# Img Test**	**Tiempo Total [s]**	**Latencia Total [ms]**	**Latencia Carga imagen [ms]**	**Latencia CNN [ms]**	**Latencia label [ms]**
16x16 px	DPU (tmpfs)	1	46	1858	33678	18,11	2,65	1,16	0,82	0,67
									Suma:	1,49

Tabla 6.4: Detalle de resultados de CPU+GPU vs. DPU.

Número de imágenes = 33.626		**Ejecución #1**	**Ejecución #2**	**Ejecución #3**	**Ejecución #4 (datos en caché)**	**Promedio**	**Desviación típica**
CPU + GPU con memoria SSD	Latencia Carga Imagen [s]	0,000495	0,009399	0,008494	0,000497	0,00612942	0,00490033
	Latencia Predicción [s]	3,515850	3,313571	3,601024	1,064921	3,47681514	0,14764845
	Latencia Total [s]	3,516345	3,322971	3,609518	1,065418	3,48294457	0,14616454
	Tiempo Total [s]	196,882371	197,549983	165,771612	13,684466	186,734655	18,1575965
	FPS	170,792336	170,215150	202,845346	2457,238718	181,284278	18,6746632
DPU con memoria SD	Latencia Carga Imagen [s]	0,025799	0,026860	0,011339	0,001249	0,02133306	0,00867099
	Latencia Predicción [s]	0,001546	0,001564	0,001554	0,002694	0,00155457	8,9854E-06
	Latencia Total [s]	0,027345	0,028425	0,012893	0,000719	0,02288763	0,00867243
	Tiempo Total [s]	38,037966	38,037966	37,796128	22,327965	37,957353	0,13962508
	FPS	884,011527	884,011527	889,667856	1506,003792	885,896970	3,26568332
DPU con memoria TMPFS	Latencia Carga Imagen [s]	0,002467	0,002390	0,002174	0,002009	0,002344	0,00015223
	Latencia Predicción [s]	0,001581	0,001642	0,001789	0,001755	0,001671	0,00010676
	Latencia Total [s]	0,004049	0,004032	0,003963	0,003764	0,004014	4,5644E-05
	Tiempo Total [s]	22,256405	22,178338	22,147577	22,199694	22,195503	0,0458918
	FPS	1510,845987	1516,164086	1518,269951	1514,705569	1514,9964	3,12997
						(no incluye la caché)	

cuales 3,511 segundos pertenecen al procesamiento de la red neuronal. En cambio, usando lote de 64 imágenes, se obtiene una tasa de 181 imágenes por segundo, la latencia total por *batch* es de 3,609 s. Por último, la tasa en el sistema DPU es de 885 imágenes, más rápido que el sistema por lotes de la GPU. Además, en la latencia por imágenes encontramos la ventaja de este sistema, pues es capaz de procesar cada imagen individualmente en 1,55 ms mientras que el sistema en lotes debe procesar 64 imágenes en 3,601 s, por lo tanto, cada paquete tendrá una latencia de procesamiento individual de 56 ms, pero estos valores no son separables deberán procesarse todos los paquetes y también habría que añadir el tiempo entre llegadas de paquetes para llenar el lote.

Si se detallan los tiempos de procesamiento en la DPU se observa el problema de cargar las imágenes con SD. Para cargar las imágenes se necesitan hasta 21 ms, mientras que el tiempo de procesamiento del modelo es de 0,87 ms, inferior a 1 ms. El tiempo de cálculo de la clase se hace mediante *arrays* que se deben procesar en la CPU por lo que

tampoco es muy eficiente, obtiene un resultado de 0,68 ms. En total, procesar la imagen cuesta 1,55 ms de los 22,88 ms totales. Para el sistema GPU en cambio se tiene la ventaja de cargar imágenes desde un disco duro SSD, por lo tanto, carga 64 imágenes en 0,49 ms y en cambio tarda 3,515 segundos en procesar la red neuronal.

Como se ha observado, los resultados de la FPGA podrían ser mejorables en un sistema real en el que no se depende de guardar imágenes en una tarjeta SD, cuyos tiempos de acceso y protocolos de comunicaciones no la hacen apta para la aplicación, además de que se optaría por un proceso al vuelo. Como último desarrollo en este proyecto se propone comparar los valores cargando las imágenes en un disco virtual, mediante el sistema de archivos `tmpfs`. La **Tabla 6.5** muestra que la carga de las imágenes mejora en un factor de 10x. Con ello, el tiempo total de cargar y procesar una imagen baja de 30 ms a 4 ms. Como se observa en la **Tabla 6.3** con este sistema se ha probado a ejecutar un conjunto de datos de 33.626 imágenes, se procesan en 22,20 segundos y se obtiene un valor de imágenes por segundo de 1515, mejorando considerablemente los valores del sistema que carga imágenes desde la memoria SD.

Para realizar este ejemplo en Windows se ha ejecutado dos veces el modelo con los mismos datos, de esta forma se puede aprovechar las políticas de cacheo del sistema y probar el rendimiento sin limitación por la SSD. En este caso, mejora la tasa de procesamiento a 2457 FPS mientras que cacheando en el sistema DPU solo se obtienen 1506 FPS, muy parecido al sistema `tmpfs` pero por debajo de la velocidad del sistema CPU+GPU. En cualquier caso la clara ventaja del sistema DPU es la baja latencia de procesamiento que se obtiene, que es independiente del sistema de memoria y además es determinista. Con este ejemplo no se pretende establecer un sistema que guarde las imágenes en SD, SSD, tmpfs o caché. Lo que se busca es demostrar que el rendimiento actualmente queda limitado debido a la carga de las imágenes y no al procesamiento en la red neuronal. Además, en un sistema de procesamiento al vuelo, estos valores de carga de imágenes no debería aplicar pues el paquete de tráfico se ejecutaría directamente en el procesador sin ser guardado en memoria. Tampoco se pretende comparar el uso de tmpfs con caché, si no que se busca el sistema de carga de imágenes más rápido disponible en cada sistema operativo.

Tabla 6.5: Resultados de latencia en carga de imágenes con `Tmpfs` vs. SD.

	Tmpfs	**SD**
Tiempo carga imagen	2.4	29
Tiempo proceso CNN	0.9	0.9
Tiempo proceso calculo clase con softmax	0.7	0.7
Tiempo Total	4	30

(todos los datos en ms)

Por último, se ha ejecutado un modelo de red neuronal con entrada para imágenes de 16x16 píxeles para comprobar si hay mejora de velocidad al utilizar imágenes más pequeñas. Los resultados obtenidos muestran que hay una mejora en la velocidad de carga de la imagen en 4 ms, es decir, 12.66 %, en el procesamiento de la red neuronal se obtiene un valor de 1.49 ms, o sea, 0,18 ms menos por imagen que es equivalente a un 9.27 % de mejora de velocidad con respecto a las imágenes de 32x32 píxeles. Este sistema

demuestra que hay una mejora en el rendimiento con las imágenes más pequeñas, pero a cambio se pierde precisión en el modelo, que desciende a 46 %. Además, se esperaría que la mejora de rendimiento fuese mayor pues las convoluciones de imágenes de 32x32 píxeles son 4 veces más pesadas respecto a las de 16x16 píxeles.

6.5. Conclusiones

En esta sección, se han comprobado los resultados del clasificador de tráfico en distintos escenarios. Se han realizado medidas de rendimiento para distintos tamaños de imágenes, según las dimensiones de estas. Se ha comprobado también el efecto de agrupar los datos según el flujo de los paquetes de tráfico utilizado, a su vez se ha demostrado que no separar por flujos obtiene valores de precisión mucho mayores.

Este resultado es muy interesante porque indica que si no se separan en flujos los datos de entrenamiento y testeo, el modelo estará resultando en valores que no se verán en una red de tráfico real, pues en estas los flujos quedan separados y no serán los mismos datos que los utilizados en el entrenamiento. También se ha probado a desordenar los flujos para comprobar el efecto en las redes LSTM y no se ha obtenido mejora o perdida de precisión. En general el modelo LSTM probado no ha resultado en valores mejores que el modelo CNN, lo que puede implicar que la información temporal que proporcionan los flujos de paquetes, no dan información extra al modelo o se pierde al mezclar entre varios flujos.

Por último, se ha comprobado qué resultado da el clasificador de tráfico en un entorno FPGA Ultrascale+ Zynq ZCU104 mediante la DPU Pynq. Los resultados en tasa de procesamiento son superiores a los obtenidos en el sistema CPU + GPU. Este sistema procesaba 181 imágenes por segundo frente al de DPU que procesaba 885 imágenes por segundo. Estos resultados pueden haber quedado limitados debido a que los sistemas DPU están diseñados para procesamiento de vídeos e imágenes más grandes, en un sistema que requiere baja latencia y alta tasa para múltiples paquetes de tráfico no este optimizado. En cuanto a los resultados del sistema DPU, se obtienen valores de latencia menores a 1 ms en el procesamiento de un paquete en la red neuronal, en el sistema CPU+GPU sube hasta los 3 segundos porque se debe procesar en lotes de 64 imágenes. El valor de latencia de la DPU es significativo pues permitiría clasificar el tráfico sin deteriorar la experiencia percibida por un usuario. Sin embargo, las pruebas realizadas quedan limitadas por la carga de las imágenes desde la memoria SD de la FPGA, al cambiar a un modelo de disco virtual en RAM mediante `tmpfs`, se obtienen valores más cercanos al que se podría esperar en un sistema al vuelo, con valores de carga por imagen cercano a 2 ms. Por último, se ha comprobado que reducir el tamaño de las imágenes obtiene una mejora de tiempos de procesamiento, a cambio se perderá precisión en el modelo.

En la web de Xilinx [61] [62] hay una comparación del rendimiento de distintas tarjetas usando distintos modelos de redes neuronales. Para el modelo *face-quality*, que es el que tiene menor *input size* y por ende el más parecido al caso planteado en este proyecto. La FPGA Zynq Ultrascale+ ZCU104 obtiene un rendimiento de 6612 FPS en procesamiento multi hilo, en cambio, para la FPGA Alveo U280 se obtiene un rendimiento de 23.735 FPS, es decir, 3,59 veces más rápida. Con estos valores, se podría calcular cuanto puede escalar la red neuronal planteada en este proyecto en el caso de utilizar una FPGA de alto rendimiento.

Con los resultados obtenidos en este trabajo se obtenía una tasa de 1521 FPS, que en una FPGA de tipo Alveo debería aumentar a 5460 FPS. En la **Figura 5.6** se observa que el tamaño medio de los paquetes procesados en este proyecto es de 1400 a 1500 bytes, recordando que los paquetes pequeños de tipo ACK han sido filtrados. Si suponemos 1450 bytes de media, se puede calcular que la tasa procesamiento de este modelo que en una FPGA Alveo sería de 5460x1450x8 = 63.34 Mbps. Para poder implementar este sistema en una red real de 10 Gbps, harían falta 158 FPGAs Alveo con el modelo desarrollado. Este resultado puede parecer inalcanzable en la actualidad, pero si se considera el aumento exponencial de núcleos y la capacidad de procesamiento de los mismos que han tenido los sistemas GPUs [63] en la última década podría resultar una opción viable en un futuro cercano.

Con estos resultados se da por completado el proyecto pues se han cumplido los requisitos y los objetivos propuestos. En el siguiente capítulo se expondrán las conclusiones obtenidas y las posibles líneas de trabajo futuro derivadas de este proyecto.

7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

En este Trabajo Fin de Máster se ha desarrollado un sistema de clasificación de tráfico. Primero, se han estudiado los sistemas de clasificación de tráfico clásicos. En segundo lugar se ha estudiado el problema de la clasificación de tráfico cifrado y cómo lo solucionan los sistemas basados en aprendizaje automático y redes neuronales. Después, se ha realizado la implementación de un sistema que convierte los paquetes de tráfico en imágenes. Estas son procesadas por un modelo de red neuronal convolucional y recurrente para predecir una clase de tráfico. Por último, se ha implementado dicho sistema en una FPGA para poder clasificar el tráfico en tiempo real.

Los resultados de precisión obtenidos son buenos, pero por debajo de los resultados obtenidos por sistemas similares en la literatura existente. Esto se debe a que el sistema de datos que se ha llevado a cabo está basado en diferenciar flujos de entrenamiento y test y condiciona mucho más de lo esperado los resultados, lo que implica que si se desea implementar un modelo de clasificación en una red real, se deberá tener en cuenta que no se pueden entrenar y testear con datos del mismo flujo, pues de otra forma se estarán obteniendo resultado que no se podrán observar en una red real.

Esto es algo en lo que fallan los artículos actuales sobre el tema, pues no establecen una metodología clara en el proceso de generación del conjunto de datos de entrenamiento y test. A la vista de nuestros resultados, los trabajos previos probablemente estén cayendo en el error de asumir que tomar los paquetes sin tener en cuenta los flujos a los que pertenecen no afecta a la precisión del modelo. Con este proyecto queda demostrado que es una cuestión muy relevante de cara a establecer un clasificador de tráfico en redes reales y se debería tener en cuenta en futuros estudios al respecto.

También se ha probado un modelo que incluye capas LSTM, para obtener información temporal de los flujos de paquetes, pero no mejoran significativamente los resultados. Parece que la información temporal de los flujos no es suficiente para clasificar mejor,

y un sistema de red convolucional obtiene los mismos resultados para un modelo menos complejo y más rápido.

Posteriormente, se ha trabajado en la implementación del sistema en FPGA, el sistema de adaptación de los modelos a archivo compilado ha resultado difícil. De hecho, la capa que une las capas convolucionales con las recurrentes no se ha podido implementar, por lo que se ha usado únicamente el modelo convolucional. El entorno de desarrollo de Xilinx Vitis AI también ha presentado errores en el proceso de compilación, que no se han podido solventar en la versión actual. La solución ha sido utilizar versiones previas, que han resultado más estables pero también más complejas de implementar. Al ejecutar el modelo en la DPU se han obtenido resultados de precisión ligeramente más bajos del modelo en GPU, esto es habitual en sistemas empotrados y se debe al proceso de cuantización de la red neuronal.

Por último, se han comprobado los resultados de tasa de procesamiento y latencia, en el primer caso se obtienen resultados ligeramente peores que en GPU, pero la latencia es hasta 100 veces menor. Esta es la clara ventaja del sistema FPGA frente al sistema clásico. Al evitar procesar las imágenes por lotes se consiguen procesar en la red neuronal las imágenes en menos de 1 ms, lo que permite analizar paquetes en tiempo real. Ha resultado difícil trabajar con una tarjeta de memoria SD, pues tiene una velocidad de carga de imágenes que empeora sustancialmente los resultados. Como esta tarjeta no debería estar en un sistema de clasificación al vuelo, se ha probado a guardar las imágenes en un disco virtual en RAM con `tmpfs`. Los resultados de carga de imágenes mejoran en un factor de 10x y se obtiene una tasa de procesamiento más cercana a lo que debería ser un sistema de procesamiento al vuelo. En las conclusiones del capítulo de resultados 6 se presenta un ejemplo de cómo se podría escalar el sistema para adaptarlo a redes de alta tasa con arquitecturas FPGA de alto rendimiento.

Para el desarrollo de este proyecto se han requerido los conocimientos de varias asignaturas del Máster, como son Planificación de Redes, Gestión de Redes, Tecnologías y Servicios de Internet, Procesamiento Avanzado de Señales Multimedia y Sistemas Electrónicos Integrados. En consecuencia a la realización del trabajo se han aumentado los conocimientos en las ramas de telemática, sistemas electrónicos y aprendizaje automático. Esto se debe al estudio de los clasificadores de red, a los sistemas de clasificación mediante redes neuronales y a los sistemas de procesamiento profundo en FPGAs.

7.2. Trabajo futuro

Si bien el desarrollo del sistema propuesto ha cumplido con los objetivos propuestos, se definen a continuación algunas de las líneas de trabajo futuro a partir de este proyecto.

1. **Desarrollar el sistema completo.** En este trabajo, se han desarrollado cada una de las partes del sistema de clasificación por separado. Se propone realizar el sistema de procesamiento en la misma FPGA, junto al procesador, de forma que se pueda conectar a la red y clasificar paquetes en tiempo real. Esto incluiría la captura del tráfico, el procesamiento en imágenes, el procesamiento con el modelo de red neuronal y un sistema de política de tráfico según la clasificación obtenida.

2. **Mejorar el sistema FPGA.** Si bien los resultados de latencia obtenidos son coherentes con el funcionamiento previsto, la tasa de procesamiento de paquetes ha resultado algo inferior a lo esperado, y no serviría, por ejemplo, para implementar en una red de alta tasa real de varios Gbit/s. Por ese motivo, se propone seguir investigando al respecto para obtener mayores velocidades. Algunas de las ideas que no se han podido desarrollar son:

   a) Añadir una segunda DPU al sistema FPGA, ya que el porcentaje de transistores utilizados es menor a la mitad. Esto permitiría duplicar la tasa efectiva. [64]

   b) También se debería valorar utilizar una FPGA más grande, como una ALVEO U200, U250, U280, o Versal, o bien usar varias FPGAs en paralelo.

   c) Buscar otras arquitecturas DPU que no estén basadas en Tensorflow, ya que aunque sea una biblioteca de Python muy utilizada en ML y fácil de prototipar, tiene un peor rendimiento comparado con Caffe o Pytorch.

3. **Usar arquitecturas de Redes Neuronales más transparentes** que permitan obtener información de cómo ven y procesan las mismas. Así, se podría ayudar a mejorar los modelos y entender por qué no han funcionado las capas LSTM como se esperaba.

4. **Comparar con un sistema de clasificación por estadísticas de tráfico.** Relacionado con lo anterior, conviene comprobar que lo que ve el modelo en las imágenes no es únicamente el tamaño del paquete y la parte cifrada no la procesan. Si fuese así, un sistema de estadísticas de flujo o paquete podría resultar más eficiente. Otra opción es cambiar el contenido del paquete a bytes aleatorios y comprobar si se obtiene una precisión parecida.

Bibliografía

[1] L. S. S. Provider, "What is deep packet inspection?." https://www.lionic.com/dpi/, Julio 2021.

[2] "Dissecting a network packet." http://books.gigatux.nl/mirror/snortids/0596006616/snortids-CHP-2-SECT-2.html, Julio 2019.

[3] F. Liu, X. Wu, W. Li, and X. Liu, "The packet size distribution patterns of the typical internet applications," *2012 3rd IEEE International Conference on Network Infrastructure and Digital Content, Beijing, China*, Septiembre 2012.

[4] D. Asanka, "Implement artificial neural networks (anns) in sql server." https://www.sqlshack.com/implement-artificial-neural-networks-anns-in-sql-server/, Abril 2020.

[5] R. Prabhu, "Understanding of convolutional neural network (cnn) — deep learning." https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148, Marzo 2018.

[6] "Training, validation, and test sets." https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets, Mayo 2021.

[7] J. Yang, J. Xu, X. Zhang, C. Wu, T. Lin, and Y. Ying, "Deep learning for vibrational spectral analysis: Recent progress and a practical guide," *Analytica Chimica Acta*, Enero 2019.

[8] B. Akera, "Alexnet: A brief review." https://medium.com/ai-research-lab-kampala/alexnet-a-brief-review-14979ce7cc84, Agosto 2018.

[9] J. I. Garzón, "Cómo usar redes neuronales (lstm) en la predicción de averías en las máquinas." https://blog.gft.com/es/2018/11/06/como-usar-redes-neuronales-lstm-en-la-prediccion-de-averias-en-las-maquinas/, Noviembre 2018.

[10] Xilinx, "Zynq ultrascale+ mpsoc zcu104 evaluation kit." https://www.xilinx.com/products/boards-and-kits/zcu104.html, Enero 2021.

[11] Xilinx, "Vitis ai user guide." https://www.xilinx.com/html_docs/vitis_ai/1_1/index.html, Febrero 2021.

[12] Xilinx, "Adaptable and real-time ai inference acceleration." `https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html`, Enero 2021.

[13] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: A novel approach for encrypted traffic classification using deep learning," *Sharif University of Technology, Teharn, Iran*, Septiembre 2017.

[14] Xilinx, "User guide zcu104 evaluation board." `https://www.manualslib.com/manual/1475689/Xilinx-Zcu104.html`, Agosto 2018.

[15] Pynq, "Zcu104 setup guide." `https://pynq.readthedocs.io/en/v2.6.1/getting_started/zcu104_setup.html`, Septiembre 2018.

[16] F. Senekal and J. Vorster, "Network mapping and usage determination," *Military Information Communications Simposium South Africa 2007, Pretoria*, Julio 2007.

[17] P. Wang, X. Chen, F. Ye, and Z. Sun, "A survey of techniques for mobile service encrypted traffic classification using deep learning," *IEEE Access*, vol. 7, pp. 54024–54033, Abril 2019.

[18] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of encrypted and vpn traffic using time-related features," *Conference: The International Conference on Information Systems Security and Privacy (ICISSP)At: ItalyVolume: 2016*, Febrero 2016.

[19] R.-H. Hwang, M.-C. Peng, V.-L. Nguyen, and Y.-L. Chang, "An lstm-based deep learning approach for classifying malicious traffic at the packet level," *Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan*, Agosto 2019.

[20] Wadulisi, "Easy ai with python and pynq." `https://www.hackster.io/wadulisi/easy-ai-with-python-and-pynq-dd4822`, Mayo 2020.

[21] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "Rfc 2475. 2.3.1 classifiers," Diciembre 1998.

[22] J. M. Berengueras, "Telefónica y netflix negocian un acuerdo de integración de contenidos." `https://www.elperiodico.com/es/economia/20180129/netflix-telefonica-movistar-acuerdo-6585176`, Enero 2018.

[23] F. Gonzalo Miguel Marín, *Deep Learning for the Analysis of Network Traffic Measurements*. Tesis, Tesis por la Universidad de la República, Facultad de Ingeniería, Marzo 2019.

[24] R. Leira Osuna, P. Gómez Nieto, I. González Vidal, and J. E. López de Vergara, *High Speed Multimedia Flow Classification*, ch. 6, pp. 93–118. John Wiley and Sons, Ltd, 2014.

[25] K. Mochalski and H. Schulze, "Deep packet inspection technology, applications and net neutrality," 2012.

[26] J. S. M. Suznejevic, "Delay limits for real-time services." `https://tools.ietf.org/id/draft-suznjevic-tsvwg-delay-limits-00.xml`, Junio 2016.

[27] M. Trevisan, D. Giordano, I. Drago, M. M. Munafò, and M. Mellia, "Five years at the edge: watching internet from the isp network," *Proceedings of the 14th International Conference on emerging Networking Experiments and Technologies*, pp. 1–12, 2018.

[28] J. L. García-Dorado and J. Aracil, "Flow-concurrence and bandwidth ratio on the internet," *COMPUTER COMMUNICATIONS, 136, 43-52, (2019)*, Enero 2019.

[29] R. APD, "¿cuáles son los tipos de algoritmos del machine learning?." `https://www.apd.es/algoritmos-del-machine-learning/`, Marzo 2019.

[30] J. V. Rueda, "Aprendizaje supervisado y no supervisado." `https://healthdataminer.com/data-mining/aprendizaje-supervisado-y-no-supervisado/`, Enero 2021.

[31] M. Merino, "Conceptos de inteligencia artificial: qué es el aprendizaje por refuerzo." `https://www.xataka.com/inteligencia-artificial/conceptos-inteligencia-artificial-que-aprendizaje-refuerzo`, Enero 2019.

[32] A. S. Khatouni, N. Seddigh, B. Nandy, and N. Zincir-Heywood, "Machine learning based classification accuracy of encrypted service channels: Analysis of various factors," *Journal of Network and Systems Management (2021) 29:8*, Marzo 2020.

[33] A. M. L. D. Guide, "Splitting the data into training and evaluation data." `https://docs.aws.amazon.com/machine-learning/latest/dg/splitting-the-data-into-training-and-evaluation-data.html`, Agosto 2016.

[34] "Estudio detallado del aa: Entrenamiento y pérdida." `https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss?hl=es`.

[35] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," *University of Science and Technology of China, Hefei, China*, Enero 2017.

[36] G. Xie, Q. Li, Y. Jiang, T. Dai, G. Shen, R. Li, R. Sinnott, and S. Xia, "Sam: Self-attention based deep learning method for online traffic classification," *Workshop on Network Meets AI and ML (NetAI'20), August 14, 2020, Virtual Event, NY, USA. ACM, New York, NY, USA,*, vol. 7, Agosto 2020.

[37] M. S. Elsayed, N.-A. Le-Khac, S. Dev, and A. D. Jurcut, "Detecting abnormal traffic in large-scale networks," *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, Octubre 2020.

[38] S. Rezaei, B. Kroencke, and X. Liu, "Large-scale mobile app identification using deep learning," *IEEE Access*, vol. PP, Diciembre 2019.

[39] V. Tong, H. A. Tran, S. Souihi, and A. Mellouk, "A novel quic traffic classifier based on convolutional neural networks," in *2018 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, 2018.

[40] A. Mahdi Sadeghzadeh, S. Shiravi, and R. Jalili, "Adversarial network traffic: Toward evaluating the robustness of deep learning based network traffic classification," Marzo 2020.

[41] E. Papadogiannaki and S. Ioannidis, "Acceleration of intrusion detection in encrypted network traffic using heterogeneous hardware," *2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Pisa, Italy*, Septiembre 2020.

[42] W. Nazih, Y. Hifny, W. S Elkilani, and T. Mostafa, "Fast detection of distributed denial of service attacks in voip networks using convolutional neural networks," *International Journal of Intelligent Computing and Information Sciences,IJICIS, Vol.20, No.2, 125-138*, 2020.

[43] "What is pynq?." `http://www.pynq.io/`.

[44] "Pynq introduction." `https://pynq.readthedocs.io/en/v2.6.1/`.

[45] Y. Zeng, H. Gu, W. Wei, and Y. Guo, "A deep learning based network encrypted traffic classification and intrusion detection framework," *IEEE Access*, vol. 7, 2019.

[46] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of encrypted and vpn traffic using time-related features," Febrero 2016.

[47] I. de Sotomonte Cañestro, *Clasificación de tráfico de red mediante redes neuronales convolucionales*. Trabajo de fin de máster, Máster Universitario en Ingeniería de Telecomunicación. Universidad Autónoma de Madrid, Junio 2020.

[48] S. Garcia-Jimenez, E. Magaña, and J. Aracil, "Natra: Network ack-based traffic reduction algorithm," *IEEE Access*, vol. 8, pp. 151229–151241, 2020.

[49] V. Morales Gómez, *Desarrollo de un Disector de Tráfico para el Protocolo QUIC*. Trabajo de fin de grado, Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación. Universidad Autónoma de Madrid, Noviembre 2019.

[50] K. Ismailaj, M. Camelo, and S. Latré, "When deep learning may not be the right tool for traffic classification," Abril 2021.

[51] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport draft-ietf-quic-transport-22." `https://tools.ietf.org/html/draft-ietf-quic-transport-22`, Julio 2019. Último acceso: 14/10/2019.

[52] "Non-alphanumeric list order from os.listdir()." `https://stackoverflow.com/questions/4813061/non-alphanumeric-list-order-from-os-listdir`, Diciembre 2017.

[53] E. Luis Bisbé, *Detección de Escenas de Violencia con Modelos Deep Learning*. Trabajo de fin de máster, Máster Universitario en Ingeniería de Telecomunicación. Universidad Autónoma de Madrid, Junio 2020.

[54] Kary, "How can the packet size be greater than the mtu?." `https://packetbomb.com/how-can-the-packet-size-be-greater-than-the-mtu/`, Agosto 2014.

[55] A. Swriski, "Vitis ai using tensorflow and keras tutorial part 7." `https://beetlebox.org/vitis-ai-using-tensorflow-and-keras-tutorial-part-7/`, Agosto 2020.

[56] "Guardar y cargar modelos." `https://stackoverflow.com/questions/59511273/8-bit-quantized-model-showing-better-train-accuracy-than-full-precision-model`.

[57] "8-bit quantized model showing better train accuracy than full precision model." `https://www.tensorflow.org/tutorials/keras/save_and_load`, Enero 2020.

[58] Y. Qu, "Training a cnn for dpu compilation." `https://github.com/Xilinx/DPU-PYNQ/blob/master/host/train_mnist_model.ipynb`, Marzo 2021.

[59] E. Mier, "Usando ramfs y tmpfs." `http://eloy-mp.com/wordpress262/usando-ramfs-y-tmpfs/`, Diciembre 2011. Último acceso: 22/12/2011.

[60] "Accuracy, precision, recall and f1 score: Interpretation of performance measures." `https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/`, Septiembre 2016.

[61] Xilinx, "Zcu104 performance." `https://www.xilinx.com/html_docs/vitis_ai/1_3/win1565723644786.html`, Junio 2021.

[62] Xilinx, "U280 performance." `https://www.xilinx.com/html_docs/vitis_ai/1_3/him1591152509554.html`, Junio 2021.

[63] K. Rupp, "Cpu, gpu and mic hardware characteristics over time." `https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`, Agosto 2016.

[64] "Dpu configuration introduction." `https://www.xilinx.com/html_docs/vitis_ai/1_3/dpu_config.html`, Junio 2021.

A

Apéndice

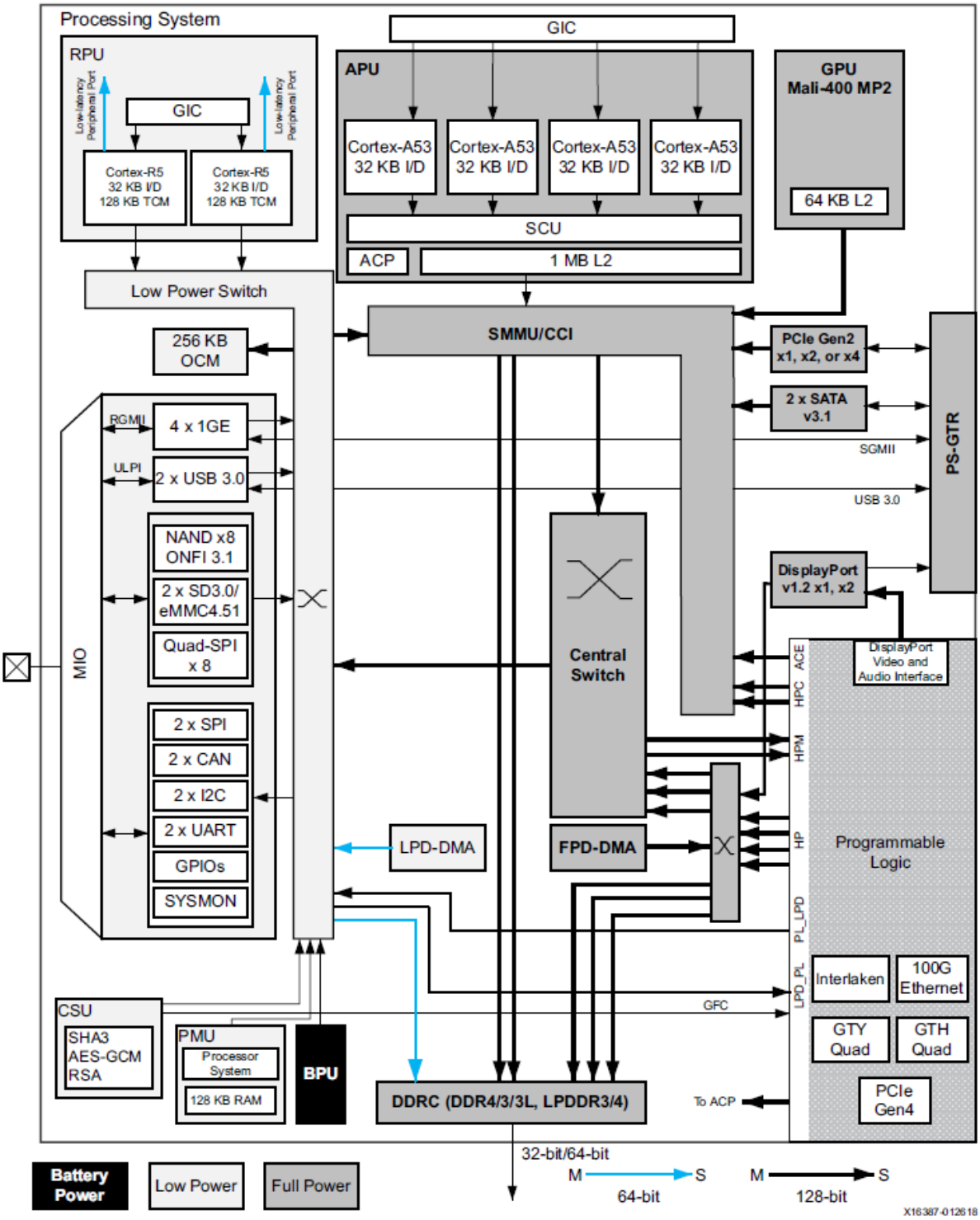

Figura A.1: Visión general de la arquitectura de la Ultrascale+ ZCU104. [14]

Table 17: **vai_q_tensorflow2 Supported Layers**

Supported Layers
tf.keras.layers.Conv2D

Table 17: **vai_q_tensorflow2 Supported Layers** *(cont'd)*

Supported Layers
tf.keras.layers.Conv2DTranspose
tf.keras.layers.DepthwiseConv2D
tf.keras.layers.Dense
tf.keras.layers.AveragePooling2D
tf.keras.layers.MaxPooling2D
tf.keras.layers.GlobalAveragePooling
tf.keras.layers.UpSampling2D
tf.keras.layers.BatchNormalization
tf.keras.layers.Concatenate
tf.keras.layers.Zeropadding2D
tf.keras.layers.Flatten
tf.keras.layers.Reshape
tf.keras.layers.ReLU
tf.keras.layers.Activation
tf.keras.layers.Add

Figura A.2: Capas Soportadas por Vitis AI Tensorflow 2. [15]

```
**************************************************
* VITIS_AI Compilation - Xilinx Inc.
**************************************************
[INFO] Namespace(inputs_shape=None, layout='NHWC', model_files=['quant
ize_results/quantized_model.h5'], model_type='tensorflow2', out_filena
me='vai_c_output/rfClassification_org.xmodel', proto=None)
[INFO] tensorflow2 model: quantize_results/quantized_model.h5
/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/site-packa
ges/xnnc/translator/tensorflow_translator.py:1809: H5pyDeprecationWarn
ing: dataset.value has been deprecated. Use dataset[()] instead.
  value = param.get(group).get(ds).value
[INFO] parse raw model     :  0%| | 0/9 [00:00<?, ?it/s]
Traceback (most recent call last):
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/bin/xnnc-run", l
ine 33, in <module>
    sys.exit(load_entry_point('xnnc==1.3.0', 'console_scripts', 'xnnc-
run')())
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/si
te-packages/xnnc/__main__.py", line 194, in main
    normal_run(args)
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/si
te-packages/xnnc/__main__.py", line 178, in normal_run
    in_shapes=in_shapes if len(in_shapes) > 0 else None,
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/si
te-packages/xnnc/xconverter.py", line 131, in run
    xmodel = CORE.make_xmodel(model_files, model_type, _layout, in_sha
pes)
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/si
te-packages/xnnc/core.py", line 104, in make_xmodel
    model_files, layout, in_shapes=in_shapes, model_type=model_t
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/si
te-packages/xnnc/translator/tensorflow_translator.py", line 97, in to_
xmodel
    model_name, raw_nodes, layout, in_shapes, model_fmt, model_type
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/si
te-packages/xnnc/translator/tensorflow_translator.py", line 163, in cr
eate_xmodel
    xmodel = cls.__create_xmodel_from_tf2(name, layers, layout, in_sha
pes)
  File "/opt/vitis_ai/conda/envs/vitis-ai-tensorflow2/lib/python3.7/si
te-packages/xnnc/translator/tensorflow_translator.py", line 381, in __
create_xmodel_from_tf2
    bottom: List[str] = [x for x in layer.get("inbound_nodes")]
TypeError: 'NoneType' object is not iterable
```

Figura A.3: Error en el proceso de compilación del modelo en VitisAI 1.3.

The `load_model()` method will automatically prepare the `graph` which is used by VART.

```
In [2]: from pynq_dpu import DpuOverlay
        overlay = DpuOverlay("dpu.bit")
        overlay.load_model("CNN_resnet.xmodel")
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-2-0a80b4d3fa64> in <module>()
      1 from pynq_dpu import DpuOverlay
      2 overlay = DpuOverlay("dpu.bit")
----> 3 overlay.load_model("CNN_resnet.xmodel")

/usr/local/lib/python3.6/dist-packages/pynq_dpu/dpu.py in load_model(self, model)
    168             self.graph = xir.Graph.deserialize(abs_model)
    169             subgraphs = get_child_subgraph_dpu(self.graph)
--> 170             assert len(subgraphs) == 1
    171             self.runner = vart.Runner.create_runner(subgraphs[0], "run")

AssertionError:
```

Figura A.4: Error en el proceso de carga del modelo en la DPU v2.6.

```
In [103]: softmax = [1, 0, 0, 0.55, 0]
          lines = ['A','B','C','D','E']
          print(np.argmax(softmax))
          print(lines[np.argmax(softmax)-1])
          print(lines[np.argmax(softmax)])
```

```
0
E
A
```

Figura A.5: Error código DPU para cálculo Softmax.

```
saver = tf.train.Saver()
tf_session = keras.backend.get_session()
input_graph_def = tf_session.graph.as_graph_def()
save_path = saver.save(tf_session, './checkpoint.ckpt')
tf.train.write_graph(input_graph_def,
                     './', 'mnist_classifier.pb', as_text=False)

nodes_names = [node.name for node in
               tf.get_default_graph().as_graph_def().node]

!freeze_graph \
    --input_graph mnist_classifier.pb \
    --input_checkpoint checkpoint.ckpt \
    --input_binary true \
    --output_graph frozen.pb \
    --output_node_names output_logits_tfm/Softmax
```

Figura A.6: Código para congelar el modelo.